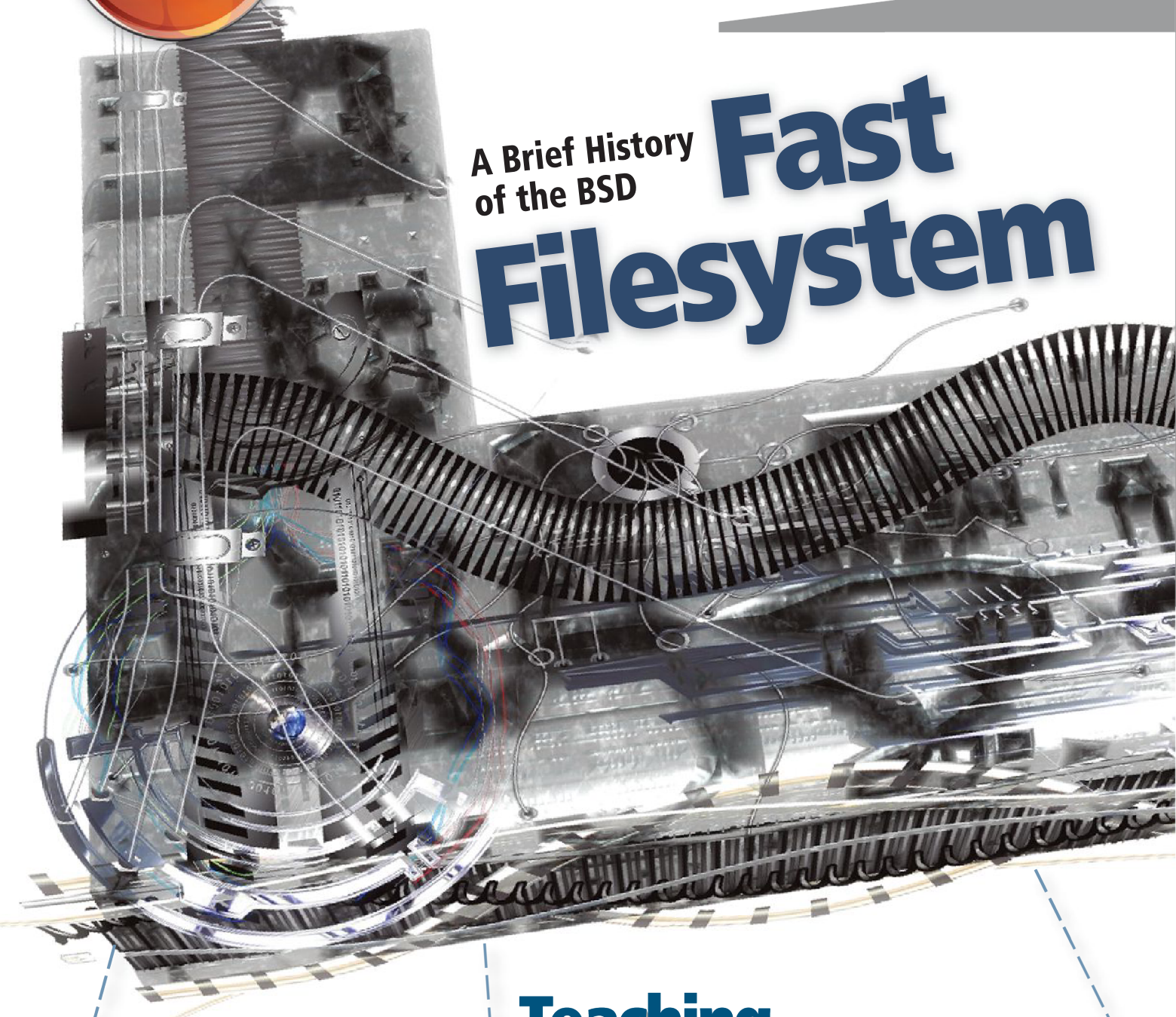
FreeBSD™ JOURNAL
March/April 2016



A Brief History of the BSD Fast Filesystem

CHERI

Building a Foundation for
SECURE, TRUSTED COMPUTING BASES

Teaching Operating Systems

with FreeBSD through Tracing, Analysis, and Experimentation

Introducing the new
XG-2758 1U pfSense® Security Appliance

Fast 10 Gigabit networking at a price you can afford.

XG-2758 1U features include:

- 8 Core Intel® Atom™ C2758 2.4 GHz with AES-NI and Quick Assist Technology.
- 16GB ECC RAM
- 4x 1Gb Ethernet RJ45 via Intel i354 on-chip; 1 port configurable RJ45 or SFP.
- 2x 10Gb Ethernet SFP+ via Intel 82599 Niantic.
- Optional PCIe x8 slot available for further expansion.
- Preloaded with pfSense Open Source software. No maintenance, licensing or upgrade fees.
- Flexible Configuration - firewall, LAN or WAN router, VPN appliance, DHCP Server, DNS Server, multi-WAN and high availability.
- Fully extendable with add-on software packages such as Snort®, Squid, SquidGuard, Suricata, to enable IDS/IPS, load balancing, traffic optimization, reporting and monitoring.
- Create VPNs to the Amazon Cloud easily with our with Amazon® AWS™ Wizard.

pfSense®

Shop now at the official pfSense store or authorized partners worldwide.
http://store.pfsense.org/XG-2758

FreeBSD™ JOURNAL March/April 2016

TABLE OF CONTENTS

Columns & Departments

3 Foundation Letter
Class Is Now in Session.
By George V. Neville-Neil

32 svn update The author intended to focus on some of the new features and enhancements of 10.3-RELEASE; however, he became so excited about the recent updates happening in 11-CURRENT that he decided to change this at the last minute.
By Steven Kreuzer

34 This Month in FreeBSD The author chats with Gleb Smirnoff, a member of the FreeBSD core, release engineering, and security teams and also a senior software developer at Netflix Inc.
By Dru Lavigne

37 Ports Report The level of activity during the January–February period was very high and some major ports were updated–which may require caution when upgrading.
By Frederic Culot

38 Conference Report
FOSDEM is one of the major free and noncommercial FOSS events in Europe. Every year, this event attracts more than 5,000 hackers and about 600 lectures. *By Rodrigo Osorio*

39 Meeting Report
The attendees at this first PortsCamp to be held in Taipei's HackerSpace spent most of their time understanding how to use poudriere(8).
By Marcelo Araujo

40 Book Review Joseph Kong reviews *The Algorithm Design Manual*, 2nd edition, *by Steven S. Skiena.*

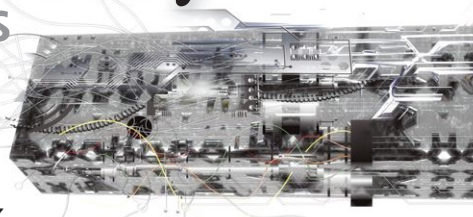
43 Events Calendar
By Dru Lavigne

TEACHING Operating Systems with FreeBSD through Tracing, Analysis, and Experimentation 4

For university operating systems courses at both the undergraduate and graduate level, the authors felt there had to be a way to use a real-world artifact such as FreeBSD, while making sure students didn't get lost in millions of lines of code.
By George V. Neville-Neil and Robert N. M. Watson

A Brief History of Fast Filesystems 12

A taxonomy of filesystem and storage development from 1979 to the present, with the BSD Fast Filesystem as its focus. **By Marshall Kirk McKusick**



CHERI 24

Building a Foundation for Secure, Trusted Computing Bases

Developed under the auspices of the DARPA Clean-slate design of Resilient, Adaptive, Secure Hosts (CRASH) program, CHERI (Capability Hardware Enhanced RISC Instructions) is a hardware/software co-design project that challenges the assumptions that have been made about hardware software interfaces for the last 40-plus years. **By Brooks Davis**

SOMETHING XL
IS COMING IN APRIL

ENTERPRISE-CLASS HARDWARE, RUNNING THE WORLD'S MOST POPULAR OPEN SOURCE STORAGE OPERATING SYSTEM.

For more information on the FreeNAS Mini, visit **iXsystems.com/mini** today.

Then, check back in April for something XL...

S&W Publishing LLC
PO BOX 408, BELFAST, MAINE 04915

Publisher • Walter Andrzejewski
walter@freebsdjournal.com

Editor-at-Large • James Maurer
jmaurer@freebsdjournal.com

Art Director • Dianne M. Kischitz
dianne@freebsdjournal.com

Office Administrator • Michael Davis
davism@freebsdjournal.com

Advertising Sales • Walter Andrzejewski
walter@freebsdjournal.com
Call 888/290-9469

FreeBSD Journal (ISBN: 978-0-615-88479-0) is published 6 times a year (January/February, March/April, May/June, July/August, September/October, November/December).

Published by the FreeBSD Foundation,
PO Box 20247, Boulder, CO 80308
ph: 720/207-5142 • fax: 720/222-2350
email: info@freebsdfoundation.org

Class Is Now in Session

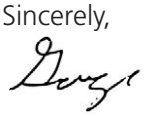
BSD got its start in the research community, having been funded, in part, by DARPA during the 1970s and '80s. As is demonstrated in this issue, FreeBSD has continued its close cooperation with research and teaching. Kirk McKusick takes us through the history of the Fast File System, the longest lived of the Unix filesystems, and the subject of continuous research and upgrading from 1979 to the present day. FFS is the de-facto standard for deployment on servers, desktops, laptops, and embedded systems. Recent research on trusted computing and capability systems at the University of Cambridge has brought us CHERI, a rethinking of computer architecture that extends the hardware with support for security features. The operating system for CHERI is FreeBSD, and the experience building this system and what it means for the future of computing are covered by Brooks Davis in "CHERI: Building a Foundation for Secure, Trusted Computing Bases." Courses on operating systems have not changed much over the 40 years that Unix and Unix-like systems have predominated within universities. The flavors have changed, but not the way in which these courses are taught. The advent of advanced tracing systems, like DTrace, a standard feature of FreeBSD, has allowed a rethinking of how OS courses are taught. Robert Watson at the University of Cambridge has been teaching a graduate level course using tracing and FreeBSD for the last two years, and I've been teaching a version for practitioners at various conferences and in industrial settings. Our experiences in building and deploying this course, which is also available online under a permissive license, are detailed in " Teaching Operating Systems with FreeBSD—Through Tracing, Analysis, and Experimentation."

The *Journal*'s columns continue to track all that's going on in FreeBSD. Dru Lavigne interviews Gleb Smirnoff, who works on FreeBSD for Netflix, as well as being a member of the FreeBSD Project's core, release engineering, and security teams. Our latest Ports Report discusses some significant changes in the ports tree which everyone should be tracking closely. And Steven Kreuzer digs into the upcoming 11 RELEASE to show us what's new and exciting in the head of the FreeBSD development branch.

Most people think of open source as occurring mostly online, but it turns out that meeting in person remains an important way of keeping a project moving along smoothly and for introducing new people to the project. Check out the report on FOSDEM by Rodrigo Osorio. FOSDEM is a huge conference with over 5,000 attendees every year.

We're well into 2016 and we've already had our first major BSD Conference, AsiaBSDCon, in Tokyo, Japan. BSDCan is coming up in June and EuroBSD in the fall. There will be a lot of chances to meet and greet with the FreeBSD world. If you can't get there we'll bring you the news here at *FreeBSD Journal*.

Sincerely,



George Neville-Neil
For the *FreeBSD Journal* Editorial Board

SEE TEXT ONLY

TEACHING Operating Systems with FreeBSD Through Tracing, Analysis, & Experimentation

By George V. Neville-Neil and Robert N. M. Watson

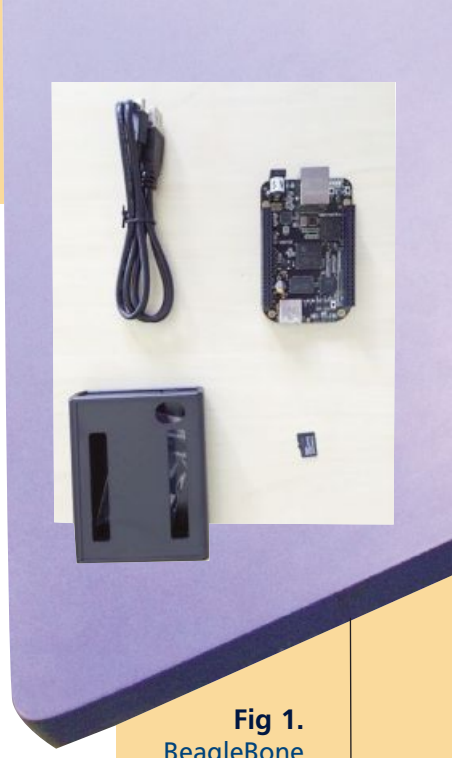


Fig 1. BeagleBone Black board used in teaching. The single USB cable is used to provide both power and communications with a lab workstation or student notebook computer.

Many people who study computer science at universities encounter their first truly large system when studying operating systems. Until their first OS course, their projects are small, self-contained, and often written by only one person or a team of three or four. Since the first courses on operating systems were begun back in the 1970s, there have been three ways in which such classes have been taught. At the undergraduate level, there is the "trial by fire," in which students extend or recreate classical elements and forms of OS design, including kernels, processes, and filesystems. In trial-by-fire courses the students are given a very large system to work with and they are expected to make small, but measurable, changes to it. Handing someone a couple million lines of C and expecting them to get something out of changing a hundred lines of it seems counterintuitive at the least. The second undergraduate style is the "toy system." With a toy system the millions of lines are reduced to some tens of thousands, which makes understanding the system as a whole easier, but severely constrains the types of problems that can be presented, and the lack of fidelity, as compared to a real, fielded operating system, often means that students do not learn a great deal about operating systems, or large systems in general. For graduate students, studying operating systems is done through a research readings course, where students read, present, discuss, and write about classic research where they are evaluated on a term project and one or more exams.

For practitioners, those who have already left the university, or those who entered computer science from other fields, there have been even fewer options. One of the few examples of a course aimed at practicing software engineers is the series "FreeBSD Kernel Internals" by Marshall Kirk McKusick, with whom both authors of this article worked on the most recent edition of *The Design and Implementation of the FreeBSD Operating System*. In the "FreeBSD Kernel Internals" courses, students are walked through the internals of the FreeBSD operating system with a generous amount of code reading and review, but without modifying the system as part of the course.

For university courses at both the undergraduate and graduate level, we felt there had to be a middle way where we could use a real-world artifact such as FreeBSD, which is deployed in products around the world, while making sure the students didn't get lost in the millions of lines of code at their disposal.

Deep-dive Experimentation

Starting in 2014, Robert and George undertook to build a "deep-dive experimentation" course for graduate students taught by Robert N. M. Watson at the University of Cambridge, as well as a practitioner course taught at conferences in industrial settings by George Neville-Neil.

In the deep-dive course, students learn about and analyze specific CPU/OS/protocol behaviors using tracing via DTrace and performance using the hwpmc(4) system. Using tracing to teach mitigates the risk of OS kernel hacking in a short course, while allowing the students to work on real-world systems rather than toys. For graduate students, we target research skills and not just OS design. The deep-dive

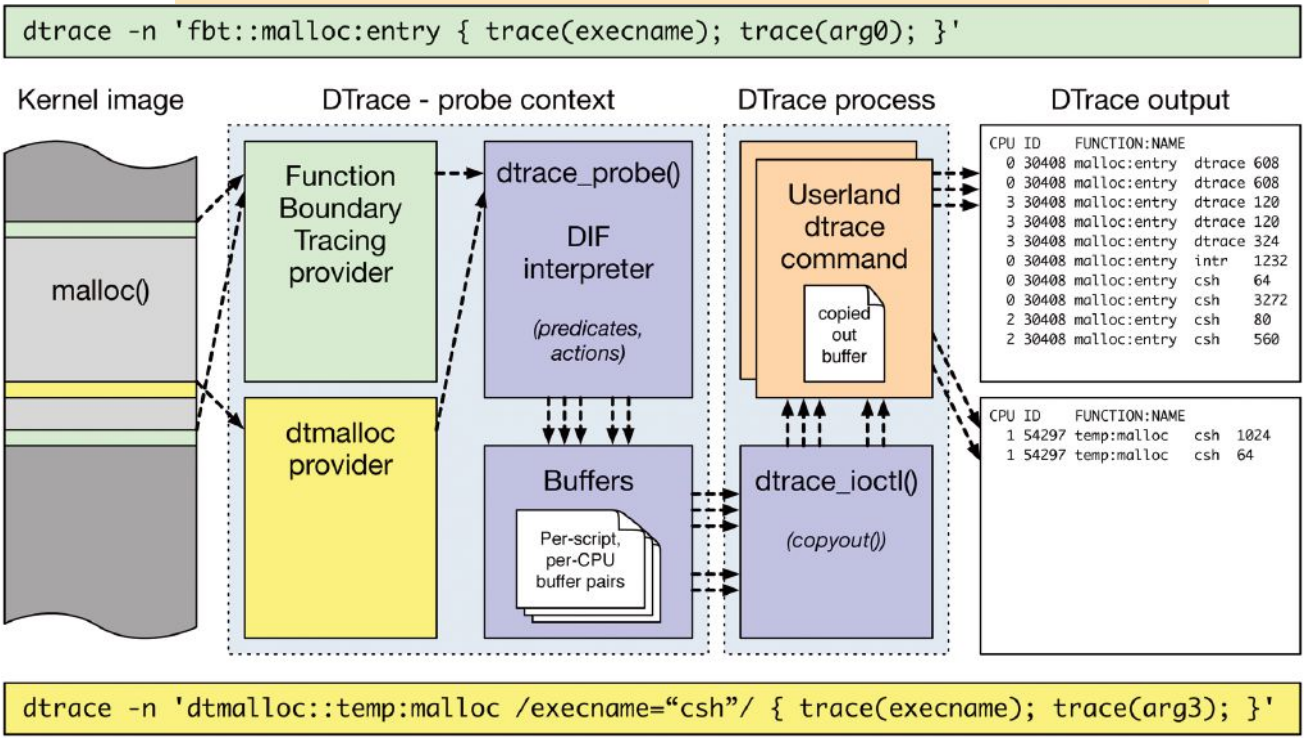


Fig. 2. DTrace is a critical part of the course's teaching approach—students trace kernels and applications to understand their performance behavior. They also need to understand—at a high level—how DTrace works, in order to reason about the "probe effect" on their measurements.

course is only possible due to development of integrated tracing and profiling tools, including DTrace and CPU performance counters present in FreeBSD.

The aims of the graduate course include teaching the methodology, skills, and knowledge required to understand and perform research on contemporary operating systems by teaching systems-analysis methodology and practice, exploring real-world systems artifacts, developing scientific writing skills, and reading selected original systems research papers.

The course is structured into a series of modules. Cambridge teaches using 8-week academic terms, providing limited teaching time compared to US-style 12-to-14-week semesters. However, students are expected to do substantial work outside of the classroom, whether in the form of reading, writing, or lab work. For the Cambridge course, we had six one-hour lectures in which we covered theory, methodology, architecture, and practice, as well as five two-hour labs. The labs included 30 minutes of extra teaching time in the form of short lectures on artifacts, tools, and practical skills. The rest of the students' time was spent doing hands-on measurement and experimentation. Readings were also assigned, as is common in graduate level courses, and these included both selected portions of module texts and historic and contemporary research papers. Students produced a series of lab reports based on experiments done in (and out) of labs. The lab reports are meant to refine scientific writing style to make it suitable for systems research. One practice run was marked, with detailed feedback given, but not assessed, while the following two reports were assessed and made up 50% of the final mark.

Three textbooks are used in the course, including *The Design and Implementation of the FreeBSD Operating System, 2nd Edition*, as the core operating systems textbook; *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*, which shows the students how to measure and evaluate their lab work; and *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD*, covering the use of the DTrace system.

Although many courses are now taught on virtual machine technology, we felt it was important to give the students experience with performance measurement. Instead of equipping a large room of

servers, we decided, instead, to teach with one of the new and inexpensive embedded boards based around the ARM series of processors. Initially we hoped to use the Raspberry Pi as it is popular, cheap, and designed at the same university at which the course would first be taught. Unfortunately, the RPi available at the time did not have proper performance counter support in hardware due to a feature being left off the system-on-chip design when it was originally produced. With the RPi out of the running, we chose the BeagleBone Black, which is built around a 1-GHz, 32-bit ARM Cortex A-8, a superscalar processor with MMU and L1/L2 caches. Each student had one of these boards on which to do lab work. The BBB has serial console and network via USB. We provided the software images on SD cards that formed the base of the students' lab work. The software images contain the FreeBSD operating system, with DTrace and support for the on-board CPU performance counters, and a set of custom micro-benchmarks. The benchmarks are used in the labs and cover areas such as POSIX I/O, POSIX IPC, and networking over TCP.

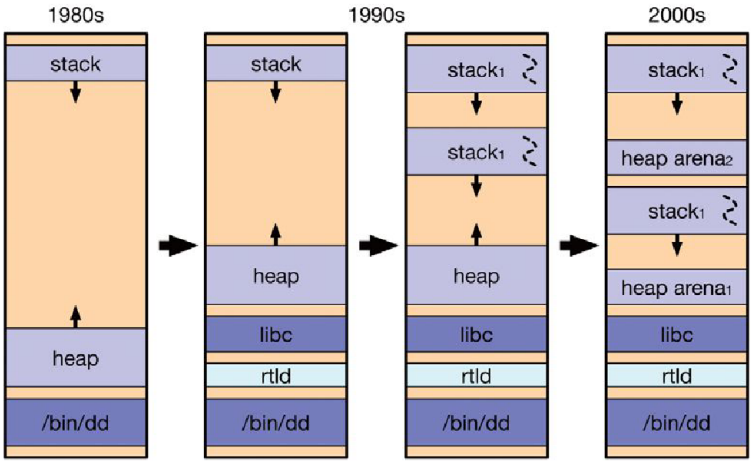


Fig 3. Students learn not just about the abstract notion of a UNIX "process," but also the evolution of the approach over the decades: dynamic linking, multithreading, and contemporary memory allocators such as FreeBSD's "jemalloc."

Eight Weeks, Three Sections

The eight weeks of the course are broken up into three major sections. In weeks one and two, there is a broad introduction to OS kernels and tracing. We want to give the students a feel for the system they are working on and the tools they'll be working with. During these first two weeks, students are assigned their first lab, in which they are expected to look at POSIX I/O performance. I/O performance is measured using a synthetic benchmark we provide in which they look at file block I/O using a constant total size with a variable buffer size. The conventional view is that increasing the buffer size will result in fewer system calls and improved overall performance, but that is not what the students will find. As buffer sizes grow, the working set first overflows the last-level cache, preventing further performance growth, and later exceeds the superpage size, measurably decreasing performance as page faults require additional memory zeroing.

The second section, covering weeks three through five, is dedicated to the process model. As the process model forms the basis of almost all modern programming systems, it is a core component of what we want the students to be able to understand and investigate during the course and afterwards in their own research. While learning about the process model, the students are also exposed to their first micro-architectural measurement lab in which they show the implications of IPC on L1 and L2 caching. The micro-architectural lab is the first one that contributes to their final grade.

The last section of the course is given over to networking, specifically the Transport Control Protocol (TCP). During weeks six through eight, the students are exposed to the TCP state machine and also measure the effects of latency on bandwidth in data transfers.

Challenges and Refinements

The graduate course has been taught twice at Cambridge, and we have reached out to other universities to talk with them about adopting the material we have produced. In teaching the course, we discovered many things that worked, as well as a few challenges to be overcome as the material is refined. We can

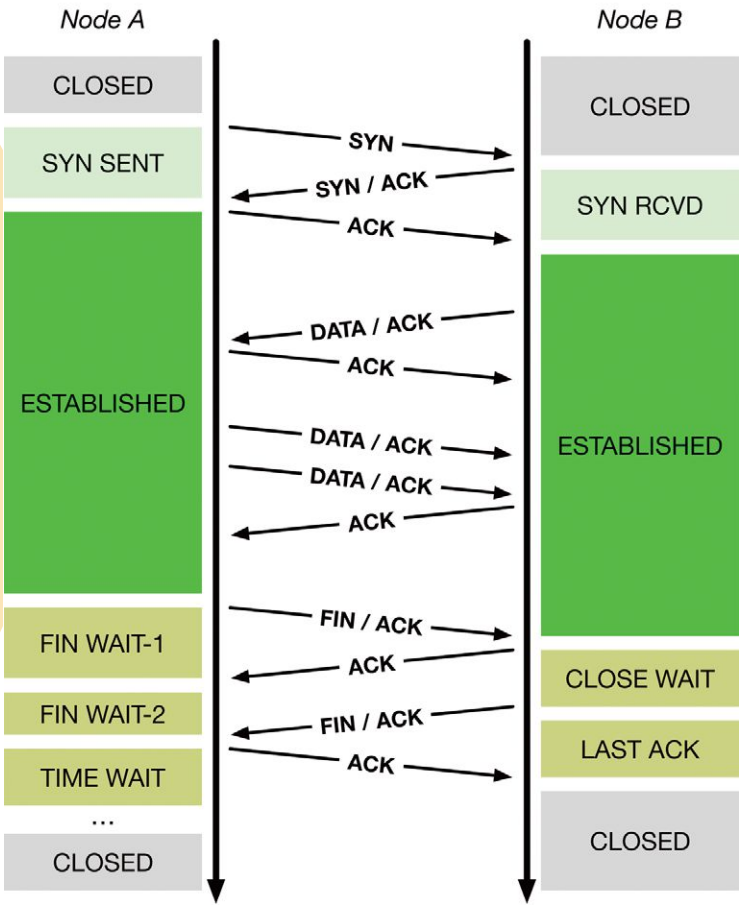


Fig 4. Labs 3 and 4 of the course require students to track the TCP state machine and congestion control using DTrace, and to simulate the effects of latency on TCP behavior using FreeBSD's DUMMYNET traffic control facility.

confirm that tracing is a great way to teach complex systems because we were able to get comprehensive and solid lab reports/analysis from the students, which was the overall goal of the course. The students were able to use cache hit vs. system-call rates to explain IPC performance. They produced TCP time-sequence plots and graphical versions of the TCP state machine all from trace output. Their lab reports had real explanations of interesting artifacts, including probe effects, superpages, DUMMYNET timer effects, and even bugs in DTrace. Our experiment with using an embedded board platform worked quite well—we could not have done most of these experiments on VMs. Overall, we found that the labs were at the right level of difficulty, but that too many experimental questions led to less focused reports—a concern addressed in the second round of teaching.

On the technical side, we should have committed to one of R, Python, or iPython Notebooks for use by the students in doing their experimental evaluations and write-ups. Having a plethora meant that there were small problems in each, all of which had to be solved and which slowed down the students' progress. When teaching the course for the first time, there were several platform bumps, including USB target issues, DTrace for ARMv7 bugs, and the 4-argument limitation for DTrace on ARMv7.

Teaching Practitioners

Teaching practitioners differs from teaching university students in several ways. First, we can assume more background, including some knowledge of programming and experience with Unix. Second, practitioners often have real problems to solve, which can lead these students to be more focused and more involved in the course work. We can't assume everything, of course, as most of the students will not have been exposed to Kernel Internals or have a deep understanding of corner cases.

Our goals for the practitioner course are to familiarize people with the tools they will use, including DTrace, and to give them practical techniques for dealing with their problems. Along the way we'll educate them about how the OS works and dispel their fears of ever understanding it. Contrary to popular belief, education is meant to dispel the students fear of a topic so that they can appreciate it more fully and learn it more deeply.

The practitioner's course is currently two eight-hour days. The platform is the student's laptop or a virtual machine. First taught at AsiaBSDCon 2015 and AsiaBSDCon 2016, the next course will be at BSDCan 2016.

Five-day, 40-hour course Hardware or VM Platform Video Recordings

Like the graduate-level course, this course is broken down into several sections and follows roughly the same narrative arc. We start by introducing DTrace using several simple and yet powerful "one liners." A DTrace one liner is a single command that yields an interesting result.

Figure below shows an example one-liner wherein every name lookup on the system is shown at run time.

```
dtrace -n 'vfs:namei:lookup:entry \
                { printf("%s", stringof(arg1));}'
       CPU     ID FUNCTION:NAME
         2  27847 lookup:entry /bin/ls
         2  27847 lookup:entry /libexec/ld-elf.so.1
         2  27847 lookup:entry /etc
         2  27847 lookup:entry /etc/libmap.conf
         2  27847 lookup:entry /etc/libmap.conf
```

The major modules are similar to the university course and cover locking, scheduler, files and the filesystem, and finally networking. The material is broken up so that each one-hour lecture is followed by a 30 minute lab in which students use the VMs on their laptops to modify examples given during the lectures or solve a directed problem. Unlike classes where we have access to hardware, the students do not take any performance measurements with hwpmc(4) since the results would be unreli-

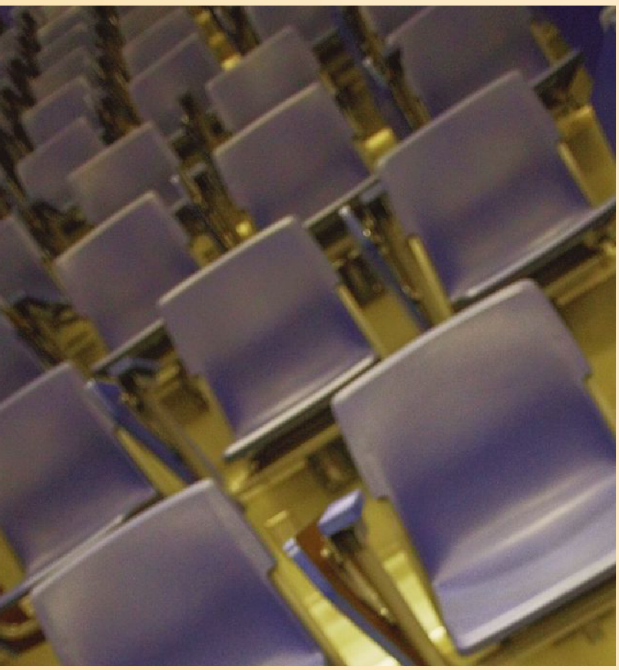
able and uninformative.

Having taught the practitioner course several times, we have learned a few things. Perhaps the most surprising was that the class really engages the students. Walking around the class during the labs, we didn't see a single person checking email or reading social media—they were actually solving the problems. The students often came up with novel answers to the problems presented, and this was only after being exposed to DTrace for a few hours. Their solutions were interesting enough that we integrated them back into the teaching during the next section. Finally, and obvious from the outset, handing a pre-built VM to the students significantly improves class startup time, with everyone focused on the task at hand, rather than tweaking their environment. Since the FreeBSD Project produces VM images for all the popular VM systems along with each release, it is easy to have the students pre-load the VM before class, or to hand them one on a USB stick when they arrive.

It's All Online!

With the overall success of these courses, we have decided to put all the material online using a permissive, BSD-like publishing license. The main page, can be found at www.teachbsd.org and our github repo, which contains all our teaching materials for both the graduate and practitioner courses can be found in github: https://github.com/teachbsd/course, where you can fork the material for your own purposes as well as send us pull requests for new features or any bugs found in the content. We would value your feedback on, and suggestions for improvements to, the course—and please let us know if you are teaching with it! ●

GEORGE V. NEVILLE-NEIL works on networking and operating system code for fun and profit. He also teaches courses on various subjects related to programming. His areas of interest are code spelunking, operating systems, networking and time protocols. He is the coauthor with Marshall Kirk McKusick and Robert N. M. Watson of *The Design and Implementation of the FreeBSD Operating System*. For over 10 years he has been the columnist better known as Kode Vicious. He earned his bachelor's degree in computer science at Northeastern University in Boston, Massachusetts, and is a member of ACM, the Usenix Association, and IEEE. He is an avid bicyclist and traveler and currently lives in New York City.



DR. ROBERT N. M. WATSON is a University Lecturer in Systems, Security, and Architecture at the University of Cambridge Computer Laboratory; FreeBSD developer and core team member; and member of the FreeBSD Foundation board of directors. He leads a number of cross-layer research projects spanning computer architecture, compilers, program analysis, program transformation, operating systems, networking, and security. Recent work includes the Capsicum security model, MAC Framework used for sandboxing in systems such as Junos and Apple iOS, and multithreading in the FreeBSD network stack. He is a coauthor of *The Design and Implementation of the FreeBSD Operating Systems (Second Edition)*.

Support FreeBSD®

Donate to the Foundation!

You already know that FreeBSD is an internationally recognized leader in providing a high-performance, secure, and stable operating system. It's because of you. Your donations have a direct impact on the Project.

Please consider making a gift to support FreeBSD for the coming year. It's only with your help that we can continue and increase our support to make FreeBSD the high-performance, secure, and reliable OS you know and love!

Your investment will help:

- Funding Projects to Advance FreeBSD
- Increasing Our FreeBSD Advocacy and Marketing Efforts
- Providing Additional Conference Resources and Travel Grants
- Continued Development of the FreeBSD Journal
- Protecting FreeBSD IP and Providing Legal Support to the Project
- Purchasing Hardware to Build and Improve FreeBSD Project Infrastructure

Making a donation is quick and easy.
freebsdfoundation.org/donate

FreeBSD™
FOUNDATION

SEE TEXT ONLY

A Brief History of the BSD FAST FILESYSTEM

by Marshall Kirk McKusick

Following is a taxonomy of filesystem and storage development from 1979 to the present, with the BSD Fast Filesystem as its focus. It describes the early performance work done by increasing the disk blocksize and by being aware of the disk geometry and using that knowledge to optimize rotational layout. With the abstraction of the geometry in the late 1980s and the ability of the hardware to cache and handle multiple requests, filesystems performance ceased trying to track geometry and instead sought to maximize performance by doing contiguous file layout. Small file performance was optimized through the use of techniques such as journaling and soft updates. By the late 1990s, filesystems had to be redesigned to handle the ever-growing disk capacities. The addition of snapshots allowed for faster and more frequent backups. Multiprocessing support got added to utilize all the CPUs found in the increasingly ubiquitous multi-core processors. The increasingly harsh environment of the Internet required greater data protection provided by access-control lists and mandatory-access controls. Ultimately, the addition of metadata optimization brings us to the present and possible future directions.

1979: Early Filesystem Work

The first work on the UNIX filesystem at University of California, Berkeley attempted to improve both the reliability and the throughput of the filesystem. The developers improved reliability by staging modifications to critical filesystem information so that the modifications could be either completed or repaired cleanly by a program after a crash [15]. Doubling the blocksize of the filesystem improved the performance of the 4.0 BSD filesystem by a factor of more than 2 when compared with the 3 BSD filesystem. This doubling caused each disk transfer to access twice as many data blocks and eliminated the need for indirect blocks for many files.

The performance improvement in the 3 BSD filesystem gave a strong indication that increasing the blocksize was a good method for improving throughput. Although the throughput had doubled, the 3 BSD filesystem was still using only about 4% of the maximum disk throughput. The main problem was that the order of blocks on the free list quickly became scrambled as files were created and removed. Eventually, the free-list order became entirely random, causing files to have their blocks allocated randomly over the disk. This randomness forced a seek before every block access. Although the 3 BSD filesystem provided transfer rates of up to 175 Kbyte per second when it was first created, the scrambling of the free list caused this rate to deteriorate to an average of 30 Kbyte per second after a few weeks of moderate use. There was no way of restoring the performance of a 3 BSD filesystem except to recreate the system.

1982: Birth of the Fast Filesystem

The first version of the current BSD filesystem was written in 1982 and became widely distributed in 4.2 BSD [14]. This version is still in use today on systems such as Solaris and Darwin. For large blocks to be used without significant waste, small files must be stored more efficiently. To increase space efficiency, the filesystem allows the division of a single filesystem block into one or more fragments. The fragment size is specified at the time that the filesystem is created; each filesystem block optionally can be broken into two, four, or eight fragments, each of which is addressable. The lower bound on the fragment size is constrained by the disk-sector size, which is typically 512 bytes. As disk space in the early 1980s was expensive and limited in size, the filesystem was initially deployed with a default blocksize of 4,096 so that small files could be stored in a single 512-byte sector.

1986: Dropping the Disk-geometry Calculations

The BSD filesystem organization divides a disk partition into one or more areas, each of which is called a cylinder group. Historically, a cylinder group comprised one or more consecutive cylinders on a disk. Although the filesystem still uses the same data structure to describe cylinder groups, the practical definition of them has changed. When the filesystem was first designed, it could get an accurate view of the disk geometry including the cylinder and track boundaries and could accurately compute the rotational location of every sector. By 1986, disks were hiding this information, providing fictitious numbers of blocks per track, tracks per cylinder, and cylinders per disk. Indeed, in modern RAID arrays, the "disk" that is presented to the filesystem may really be composed from a collection of disks in the RAID array. While some research has been done to figure out the true geometry of a disk [5, 10, 25], the complexity of using such information effectively is high. Modern disks have greater numbers of sectors per track on the outer part of the disk than the inner part that makes calculation of the rotational position of any given sector complex to calculate. So in 1986, all the rotational layout code was deprecated in favor of laying out files using numerically close block numbers (sequential being viewed as optimal) believing that would give the best performance. Although the cylinder group structure is retained, it is used only as a convenient way to manage logically close groups of blocks.

1987: Filesystem Stacking

The early vnode interface was simply an object-oriented interface to an underlying filesystem. By 1987 demand had grown for new filesystem features. It became desirable to find ways of providing them without having to modify the existing, and stable, filesystem code. One approach is to provide a mechanism for stacking several filesystems on top of one another [24]. The stacking ideas were refined and implemented in the 4.4 BSD system [7]. The bottom of a vnode stack tends to be a disk-based filesystem, whereas the layers used above it typically transform their arguments and pass on those arguments to a lower layer.

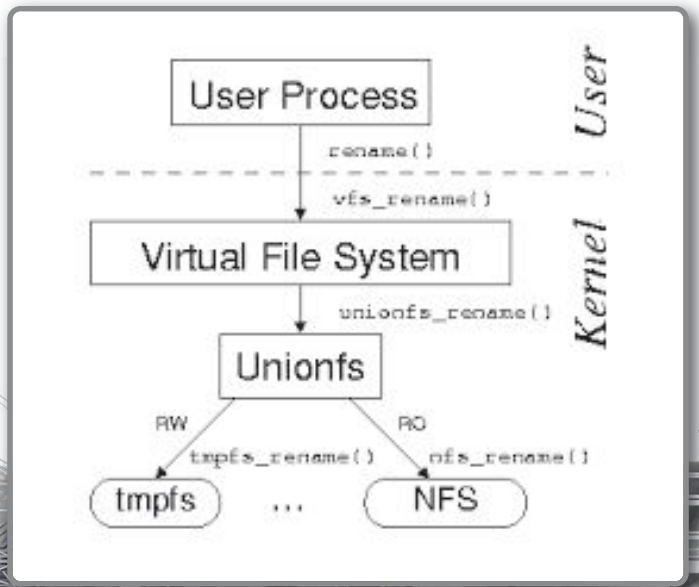
Stacking uses the mount command to create new layers. The mount command pushes a new layer onto a vnode stack; an unmount command removes a layer. Like the mounting of a filesystem, a vnode stack is visible to all processes running on the system. The mount command identifies the underlying layer in the stack, creates the new layer, and attaches that layer into the filesystem name space. The new layer can be attached to the same place as the old layer (covering the old layer) or to a different place in the tree (allowing both layers to be visible).

When a file access (e.g., an open, read, stat, or close) occurs to a vnode in the stack, that vnode has several options:

- Do the requested operations and return a result.
- Pass the operation without change to the next-lower vnode on the stack. When the operation returns from the lower vnode, it may modify the results or simply return them.
- Modify the operands provided with the request and then pass it to the next-lower vnode. When the operation returns from the lower vnode, it may modify the results, or simply return them. If an operation is passed to the bottom of the stack without any layer taking action on it, then the interface will return the error "operation not supported."

The simplest filesystem layer is nullfs. It makes no transformations on its arguments, simply passing through all requests that it receives and returning all results that it gets back. Although it provides no useful functionality if it is simply stacked on top of an existing vnode, nullfs can provide a loopback filesystem by mounting the filesystem rooted at its source vnode at some other location in the filesystem tree. The code for nullfs is also an excellent starting point for designers who want to build their own filesystem layers. Examples that could be built include a compression layer or an encryption layer.

The union filesystem is another example of a middle filesystem layer. Like the nullfs, it does not store data but just provides a name-space transformation. It is loosely modeled on the work on the 3-D filesystem [9], on the Translucent filesystem [8], and on the Automounter [20]. The union filesystem takes an existing filesystem and transparently overlays the latter on another filesystem. Unlike most other filesystems, a union mount does not cover up the directory on which the filesystem is mounted. Instead, it shows the logical merger of both directories and allows both directory trees to be accessible simultaneously [19].



1988: Raising the Blocksize

By 1988 disk capacity had risen enough that the default blocksize was raised to 8,196-byte blocks with 1,024-byte fragments. Although this meant that small files used a minimum of two disk sectors, the nearly doubled throughput provided by doubling the blocksize seemed a reasonable trade-off for the measured 1.4% of additional wasted space.

1990: Dynamic Block Reallocation

Through most of the 1980s, the optimal placement for files was to lay them out using every other block on the disk. By leaving a gap between each allocated block, the disk had time to schedule the next read or write following the completion of the previous operation. With the advent of disk caches and the ability to handle multiple outstanding requests (tag queueing) in the late 1980s, it became desirable to begin laying files out contiguously on the disk.

The operating system has no way of knowing how big a file will be when it is first opened for writing. If it assumes that all files will be big and thus tries to place them in its largest area of available space, it will soon have only small areas of contiguous space available. Conversely, if it assumes that all files will be small and thus tries to place them in its areas of fragmented space, then the beginning of files that do grow large will be poorly laid out.

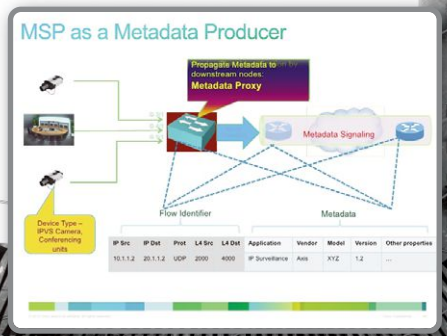
To avoid these problems, the filesystem was changed in 1990 to do dynamic block reallocation. The filesystem initially places the file's blocks in small areas of free space, but then moves them to larger areas of free space as the file grows. Using this technique, small files use the small chunks of free space while the large ones get laid out contiguously in the large areas of free space. The algorithm does not tend to increase I/O load as the buffer cache generally holds the file contents long enough that the final block allocation has been determined by the first time that the file data is flushed to disk.

The effect of this algorithm is that the free space remains largely unfragmented even after years of use. A Harvard study found only a 15% degradation in throughput on a three-year-old filesystem versus a 40% degradation on an identical filesystem that had had the dynamic reallocation disabled [27].

1996: Soft Updates

In filesystems, metadata (e.g., directories, inodes, and free block maps) gives structure to raw storage capacity. Metadata provides pointers and descriptions for linking multiple disk sectors into files and identifying those files. To be useful for persistent storage, a filesystem must maintain the integrity of its metadata in the face of unpredictable system crashes, such as power interruptions and operating system failures. Because such crashes usually result in the loss of all information in volatile main memory, the information in nonvolatile storage (i.e., disk) must always be consistent enough to deterministically reconstruct a coherent filesystem state.

Specifically, the on-disk image of the filesystem must have no dangling pointers to uninitialized space, no ambiguous resource ownership caused by multiple pointers, and no unreferenced live resources. Maintaining these invariants generally requires sequencing (or atomic grouping) of updates to small on-disk



metadata objects.

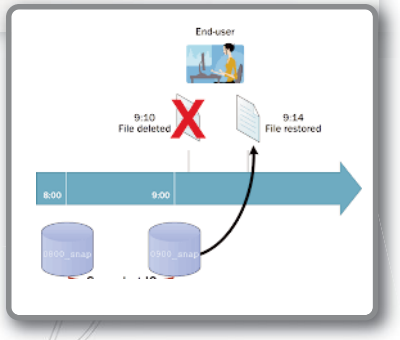
Traditionally, the filesystem used synchronous writes to properly sequence stable storage changes. For example, creating a file involves first allocating and initializing a new inode and then filling in a new directory entry to point to it. With the synchronous write approach, the filesystem forces an application that creates a file to wait for the disk write that initializes the on-disk inode. As a result, filesystem operations like file creation and deletion proceed at disk speeds rather than processor/memory speeds [16,18, 26]. Since disk access times are long compared to the speeds of other computer components, synchronous writes reduce system performance.

The metadata update problem can also be addressed with other mechanisms. For example, one can eliminate the need to keep the on-disk state consistent by using NVRAM technologies, such as an uninterruptible power supply or Flash RAM [17, 33]. Filesystem operations can proceed as soon as the block to be written is copied into the stable store, and updates can propagate to disk in any order and whenever it is convenient. If the system fails, unfinished disk operations can be completed from the stable store when the system is rebooted.

Another approach is to group each set of dependent updates as an atomic operation with some form of write-ahead logging [3, 6] or shadow-paging [2, 28]. These approaches augment the on-disk state with a log of filesystem updates on a separate disk or in stable store. Filesystem operations can then proceed as soon as the operation to be done is written into the log. If the system fails, unfinished filesystem operations can be completed from the log when the system is rebooted. Many modern filesystems successfully use write-ahead logging to improve performance compared to the synchronous write approach.

An alternative approach called soft updates was evaluated in the context of a research prototype [4]. Following a successful evaluation, a production version of soft updates was written for BSD in 1996. With soft updates, the filesystem uses delayed writes (i.e., write-back caching) for metadata changes, tracks dependencies between updates, and enforces these dependencies at write-back time. Because most metadata blocks contain many pointers, cyclic dependencies occur frequently when dependencies are recorded only at the block level. Therefore, soft updates track dependencies on a per-pointer basis, which allows blocks to be written in any order. Any still-dependent updates in a metadata block are rolled back before the block is written and rolled forward afterwards. Thus, dependency cycles are eliminated as an issue. With soft updates, applications always see the most current copies of metadata blocks, and the disk always sees copies that are consistent with its other contents.

1999: Snapshots



In 1999, the filesystem added the ability to take snapshots. A filesystem snapshot is a frozen image of a filesystem at a given instant in time. Snapshots support several important features: the ability to provide backups of the filesystem at several times during the day and the ability to do reliable dumps of live filesystems.

Snapshots may be taken at any time. When taken every few hours during the day, they allow users to retrieve a file that they wrote several hours earlier and later deleted or overwrote by mistake. Snapshots are much more convenient to use than dump tapes and can be created much more frequently.

To make a snapshot accessible to users through a traditional filesystem interface, the system administrator uses the mount command to place the replica of the frozen filesystem at whatever location in the name space that is convenient.

Once filesystem snapshots are available, it becomes possible to safely dump live filesystems. When dump notices that it is being asked to dump a mounted filesystem, it can simply take a snapshot of the filesystem and run over the snapshot instead of on the live filesystem. When dump completes, it releases the snapshot.

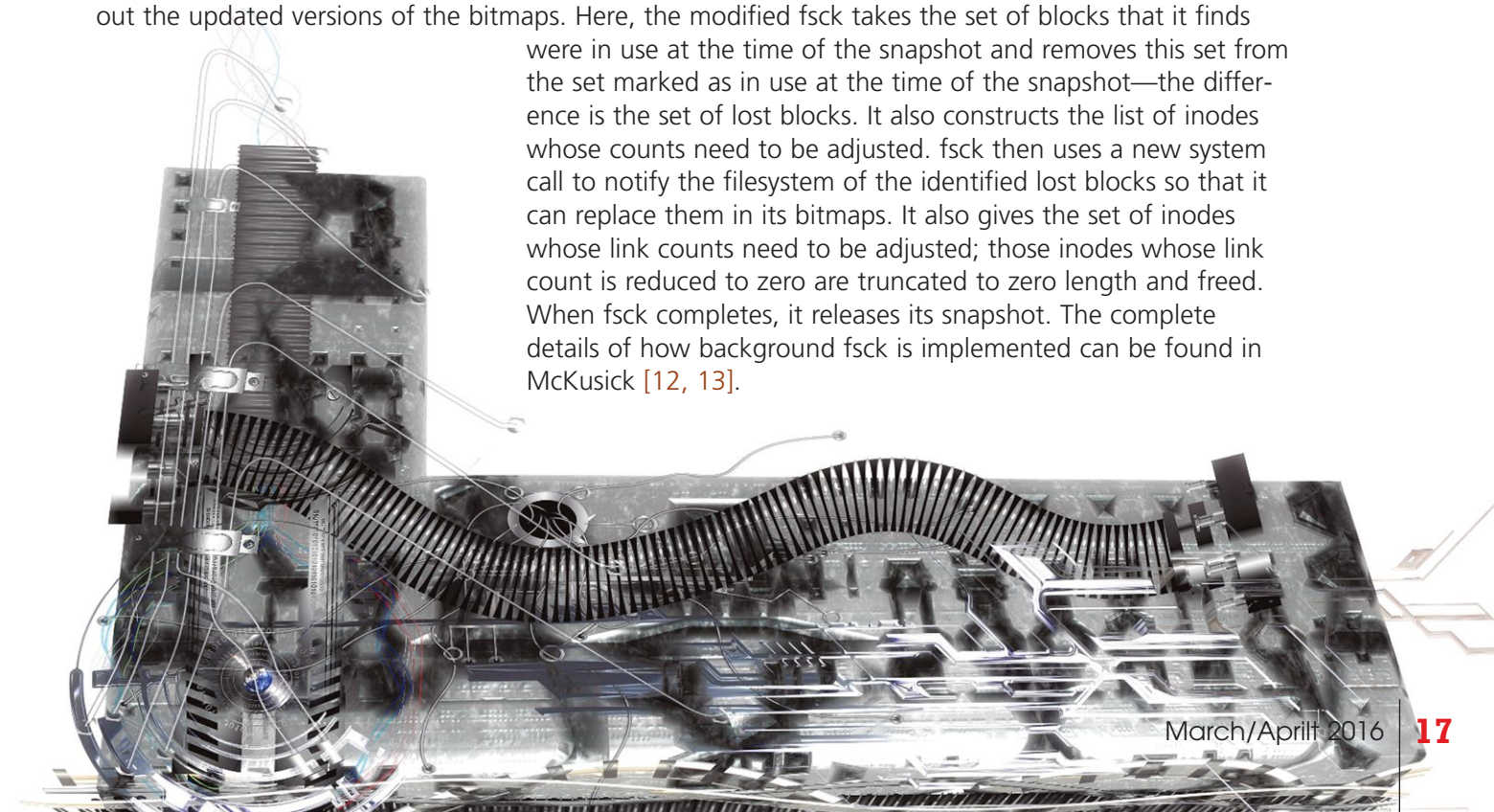
2001: Raising the Blocksize, Again

By 2001 disk capacity had risen enough that the default blocksize was raised to 16,384-byte blocks with 2,048-byte fragments. Although this meant that small files used a minimum of four disk sectors, the nearly doubled throughput provided by doubling the blocksize seemed a reasonable trade-off for the measured 2.9% of additional wasted space.

2002: Background fsck

Traditionally, after an unclean system shutdown, the filesystem check program, fsck, has had to be run over all the inodes in a filesystem to ascertain which inodes and blocks are in use and to correct the bitmaps. This check is a painfully slow process that can delay the restart of a big server for an hour or more. Soft updates guarantee the consistency of all filesystem resources, including the inode and block bitmaps. With soft updates, the only inconsistency that can arise in the filesystem (barring software bugs and media failures) is that some unreferenced blocks may not appear in the bitmaps and some inodes may have to have overly high link counts reduced. Thus, it is completely safe to begin using the filesystem after a crash without first running fsck. However, some filesystem space may be lost after each crash. Thus, there is value in having a version of fsck that can run in the background on an active filesystem to find and recover any lost blocks and adjust inodes with overly high link counts.

With the addition of snapshots, the task becomes simple, requiring only minor modifications to the standard fsck. When run in background cleanup mode, fsck starts by taking a snapshot of the filesystem to be checked. fsck then runs over the snapshot filesystem image doing its usual calculations just as in its normal operation. The only other change comes at the end of its run, when it wants to write out the updated versions of the bitmaps. Here, the modified fsck takes the set of blocks that it finds were in use at the time of the snapshot and removes this set from the set marked as in use at the time of the snapshot—the difference is the set of lost blocks. It also constructs the list of inodes whose counts need to be adjusted. fsck then uses a new system call to notify the filesystem of the identified lost blocks so that it can replace them in its bitmaps. It also gives the set of inodes whose link counts need to be adjusted; those inodes whose link count is reduced to zero are truncated to zero length and freed. When fsck completes, it releases its snapshot. The complete details of how background fsck is implemented can be found in McKusick [12, 13].



2003: Multi-terabyte Support

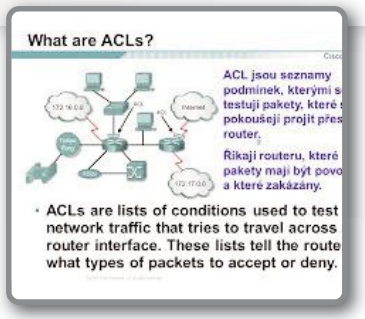
The original BSD fast filesystem and its derivatives have used 32-bit pointers to reference the blocks used by a file on the disk. At the time of its design in the early 1980s, the largest disks were 330 Mbytes. There was debate at the time whether it was worth squandering 32 bits per block pointer rather than using the 24-bit block pointers of the filesystem that it replaced. Luckily the futurist view prevailed, and the design used 32-bit block pointers.

Over the 20 years since it has been deployed, storage systems have grown to hold over a terabyte of data. Depending on the blocksize configuration, the 32-bit block pointers of the original filesystem run out of space in the 1 to 4 terabyte range. While some stopgap measures can be used to extend the maximum-size storage systems supported by the original filesystem, by 2002 it became clear the only long-term solution was to use 64-bit block pointers. Thus, we decided to build a new filesystem, that would use 64-bit block pointers.

We considered the alternatives between trying to make incremental changes to the existing filesystem versus importing another existing filesystem such as XFS [29], or ReiserFS [21]. We also considered writing a new filesystem from scratch so that we could take advantage of recent filesystem research and experience. We chose to extend the original filesystem because this approach allowed us to reuse most of its existing code base. The benefits of this decision were that the 64-bit block-based filesystem was developed and deployed quickly, it became stable and reliable rapidly, and the same code base could be used to support both 32-bit block and 64-bit block filesystem formats. Over 90% of the code base is shared, so bug fixes and feature or performance enhancements usually apply to both filesystem formats.

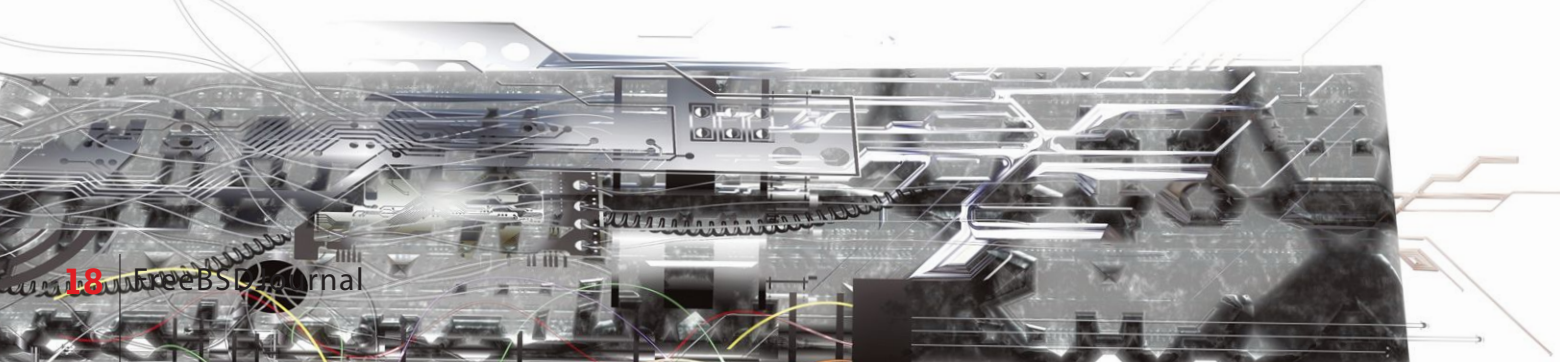
At the same time that the filesystem was updated to use 64-bit block pointers, an addition was made to support extended attributes. Extended attributes are a piece of auxiliary data storage associated with an inode that can be used to store auxiliary data that is separate from the contents of the file. The idea is similar to the concept of data forks used in the Apple filesystem [1]. By integrating the extended attributes into the inode itself, it is possible to provide the same integrity guarantees as are made for the contents of the file itself. Specifically, the successful completion of an fsync system call ensures that the file data, the extended attributes, and all names and paths leading to the names of the file are in stable store.

2004: Access Control Lists



Extended attributes were first used to support an access control list, generally referred to as an ACL. An ACL replaces the group permissions for a file with a more specific list of the users who are permitted to access the files. The ACL also includes a list of the permissions that each user is granted. These permissions include the traditional read, write, and execute permissions along with other properties such as the right to rename or delete the file [22].

Earlier implementations of ACLs were done with a single auxiliary file per filesystem that was indexed by the inode number and had a small fixed-sized area to store the ACL permissions. The small size was to keep the size of the auxiliary file reasonable, since it had to have space for every possible



inode in the filesystem. There were two problems with this implementation. The fixed size of the space per inode to store the ACL information meant that it was not possible to give access to long lists of users. The second problem was that it was difficult to atomically commit changes to the ACL list for a file, since an update required that both the file inode and the ACL file be written to have the update take effect [30].

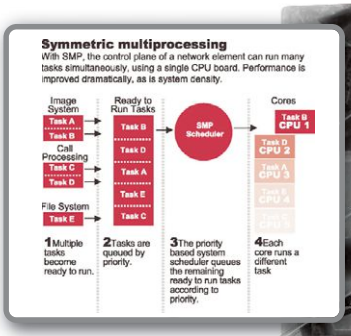
Both problems with the auxiliary file implementation of ACLs are fixed by storing the ACL information directly in the extended-attribute data area of the inode. Because of the large size of the extended attribute data area (a minimum of 8 Kbytes and typically 32 Kbytes), long lists of ACL information can be easily stored. Space used to store extended attribute information is proportional to the number of inodes with extended attributes and the size of the ACL lists that they use. Atomic update of the information is much easier, since writing the inode will update the inode attributes and the set of data that it references including the extended attributes in one disk operation. While it would be possible to update the old auxiliary file on every fsync system call done on the filesystem, the cost of doing so would be prohibitive. Here, the kernel knows if the extended attribute data block for an inode is dirty and can write just that data block during an fsync call on the inode.

2005: Mandatory Access Controls

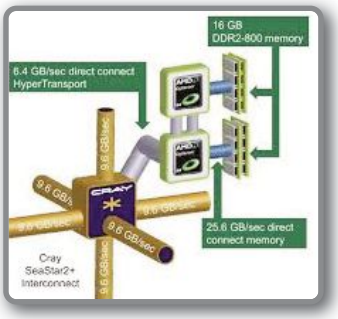
The second use for extended attributes was for data labeling. Data labels provide permissions for a mandatory access control (MAC) framework enforced by the kernel. The kernel's MAC framework permits dynamically introduced system-security modules to modify system security functionality. This framework can be used to support a variety of new security services, including traditional labeled mandatory access control models. The framework provides a series of entry points that are called by code supporting various kernel services, especially with respect to access control points and object creation. The framework then calls out to security modules to offer them the opportunity to modify security behavior at those MAC entry points. Thus, the filesystem does not codify how the labels are used or enforced. It simply stores the labels associated with the inode and produces them when a security module needs to query them to do a permission check [31, 32].

2006: Symmetric Multi-processing

In the late 1990s, the Free BSD Project began the long hard task of converting their kernel to support symmetric multi-processing. The initial step was to add a giant lock around the entire kernel to ensure that only one processor at a time could be running in the kernel. Each kernel subsystem was brought out from under the giant lock by rewriting it to be able to be executed by more than one processor at a time. The vnode interface was brought out from under the giant lock in 2004. The disk subsystem became multi-processor safe in 2005. Finally, in 2006, the fast filesystem was overhauled to support symmetric multi-processing completing the giant-free path from system-call to hardware.



2009: Journaled Soft Updates

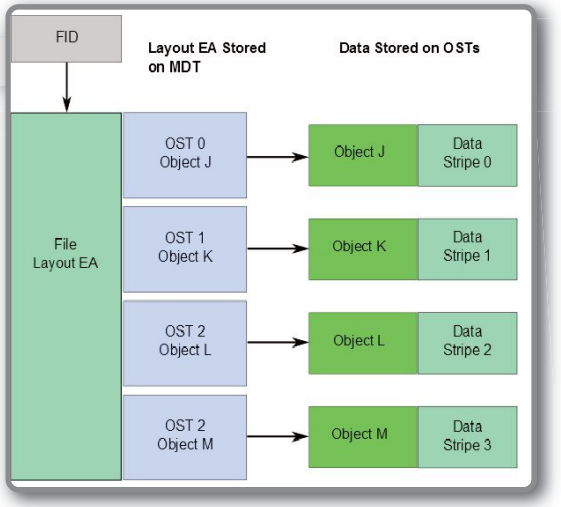


Though soft updates avoided the need to run fsck after a crash, soft updates could still cause blocks and inodes to become lost; they would not be in use by the filesystem but were still claimed as in use in the filesystem's bitmaps. fsck still had to be run periodically to reclaim the lost space. Thus, the idea arose to supplement soft updates with a journal that tracks the freeing of resources so that after a crash the journal can be replayed to recover the lost resources. Specifically, the journal contains the information needed to recover the block and inode resources that have been freed but whose freed status failed to make it to disk before a system failure. After a crash, a variant of the venerable fsck program runs through the journal to identify and free the lost resources. Only if an inconsistency between the log and filesystem is detected is it necessary to run the whole-filesystem fsck. The journal is tiny: 16 Mbyte is usually enough, independent of filesystem size. Although journal processing needs to be done before restarting, the processing time is typically just a few seconds and, in the worst case, a minute. It is not necessary to build a new filesystem to use soft-updates journaling. The addition or deletion of soft-updates journaling to existing Free BSD fast filesystems is done using the tunefs program.

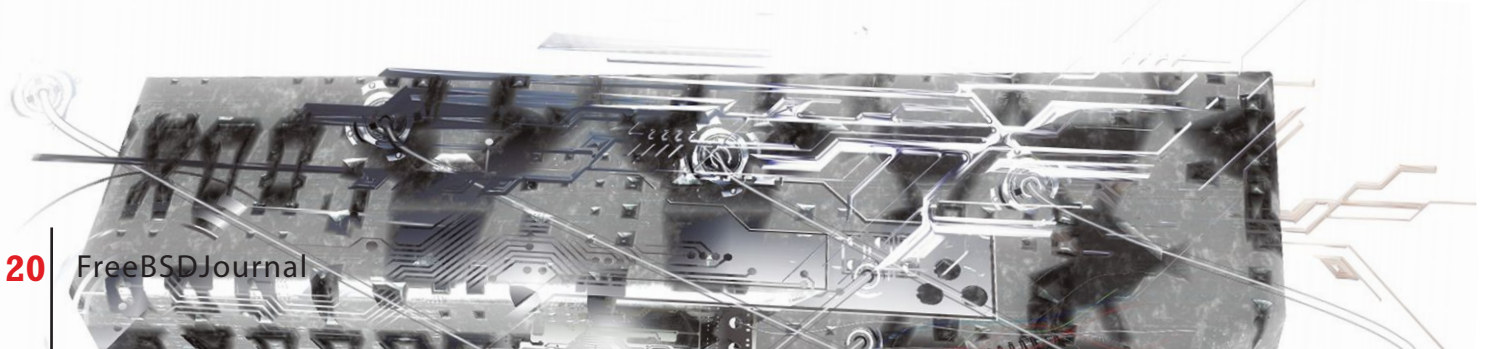
2011: Raising the Blocksize, Yet Again

By 2011 disk capacity had risen enough that the default blocksize was raised to 32,768-byte blocks with 4,096-byte fragments. This increase was also driven by the change of disk technology to 4K sectors. With the increase in sector size, small files once again used a minimum of one disk sector. Thus the filesystem once again doubled throughput with no additional wasted disk space.

2013: Optimized Metadata Layouts



In an effort to speed random access to files and to speed the checking of metadata by fsck, the filesystem holds the first 4% of the data blocks in each cylinder group for the use of metadata [11]. The policy routines preferentially place metadata in the metadata area and everything else in the blocks that follow the metadata area. The size of the metadata area does not need to be precisely calculated as it is used just as a hint of where to place the metadata by the policy routines. If the metadata area fills up, then the metadata can be placed in the regular-blocks area, and if the regular-blocks area fills up, then the regular blocks can be placed in the metadata area. This decision happens on a cylinder group by cylinder group basis, so some cylinder groups can overflow their metadata area while others do not



overflow it. The policy is to place all metadata in the same cylinder group as their inode. Spreading the metadata across cylinder groups generally results in reduced filesystem performance.

The one exception to the metadata placement policy is for the first indirect block of the file. The policy is to place the first (single) indirect block inline with the file data (e.g., it tries to lay out the first 12 direct blocks contiguously, followed immediately by the indirect block, followed immediately by the data blocks referenced from the indirect block). Putting the first indirect block inline with the data rather than in the metadata area is to avoid two extra seeks when reading it. These two extra seeks would noticeably slow down access to a file that uses only the first few blocks referenced from its indirect block.

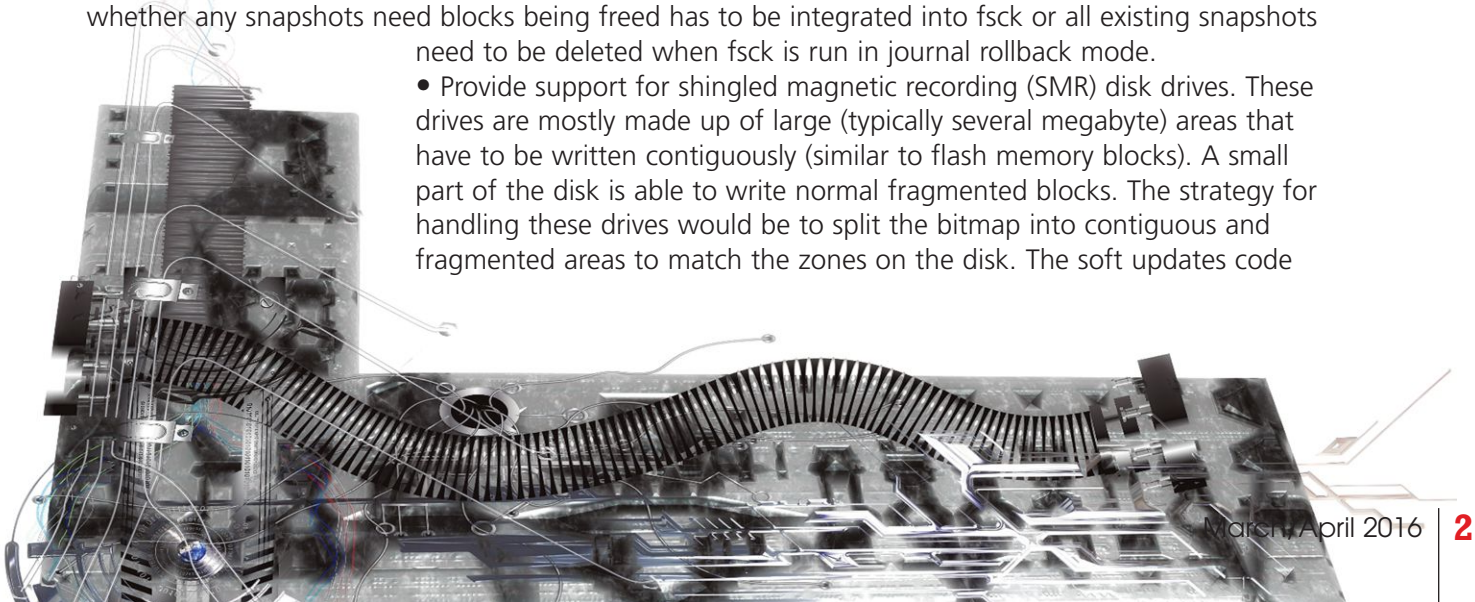
Only the second and third level indirects, along with the indirects that they reference, are allocated in the metadata area. The nearly contiguous allocation of this metadata close to the inode that references them noticeably improves the random access time to the file as well as speeding up the running time of fsck. Also, the disk track cache is often filled with much of a file's metadata when the second-level indirect block is read, thus often speeding up even the sequential reading time for the file.

In addition to putting indirect blocks in the metadata area, it is also helpful to put the blocks holding the contents of directories there, too. Putting the contents of directories in the metadata area gives a speed-up to directory tree traversal since the data is a short seek away from where the directory inode was read and may already be in the disk's track cache from other directory reads done in its cylinder group.

Future Directions

There have been many changes proposed or requested for the filesystem. The first group includes improvements that can be made without the need to change the on-disk format of the filesystem:

- When running on devices such as flash memory that need to bulk-erase blocks before they can be reused, the filesystem notifies the underlying device whenever it is finished using a block. The device is notified using a TRIM request, typically when a block is being freed during a file removal or truncation. Each block that is freed results in a TRIM command being sent through the GEOM layer to the device and a corresponding acknowledgment being returned when it is completed. Often, a file is made up of many consecutive blocks. It would be far more efficient to collect these consecutive blocks together and send a single TRIM request for the entire large block.
- Currently, a filesystem that has enabled journaled soft updates is not able to take snapshots. The restriction applies because the code in fsck that does the journal rollback does not know how to handle snapshot files when it is releasing blocks. Specifically, when a block is released, each of the snapshots needs to be checked to see if the block being released is one that the snapshot wants to claim. Only if none of the snapshots want the block can it be released to the free list. The journal rollback in fsck always releases the blocks to the free list. Thus, if the filesystem contained any snapshots that needed to claim one of the released blocks, it would be corrupted. Either the kernel code that checks whether any snapshots need blocks being freed has to be integrated into fsck or all existing snapshots need to be deleted when fsck is run in journal rollback mode.
- Provide support for shingled magnetic recording (SMR) disk drives. These drives are mostly made up of large (typically several megabyte) areas that have to be written contiguously (similar to flash memory blocks). A small part of the disk is able to write normal fragmented blocks. The strategy for handling these drives would be to split the bitmap into contiguous and fragmented areas to match the zones on the disk. The soft updates code



would then be augmented to collect the soft updates together into batches of blocks that could be written contiguously.

- The current filesystem has a provision to allow individual files to use a larger blocksize. This capability has never been implemented, but would be very beneficial for larger files. Other desired changes to the filesystem would require a new disk format, often referred to as UFS3:

  The most important of these changes would be to change the directory format to increase its file number from 32 bits to 64 bits. With increasing disk capacities, the limit of 4 billion files per filesystem has become increasingly problematic. The biggest stumbling block to making this change is the FreeBSD filesystem interface currently only supports 32-bit file numbers. Changing this interface has been discussed ever since the ZFS filesystem with its native 64-bit file numbers was brought into the system. Hopefully the interface change can be realized in time for the FreeBSD 11 release.
- Another limitation of the current filesystem format is that it uses a 16-bit field to record the number of links to a file, thus limiting a file to having 65,535 directory entries referencing it. When changing the on-disk format, this field should be increased to 64 bits to resolve the problem for many years to come.When doing a revision of the on-disk format as part of UFS3, it might be useful to add some ZFS-like features:

  Add checksums to improve the robustness and data integrity of the filesystem. The easiest to implement would be to place the checksum in each block of the file. This approach fails to detect blocks that are written to the wrong location. Taking the ZFS approach of storing each checksum with the block pointer avoids this problem but requires a more impactful change to the existing filesystem code.
- Another useful change to improve the robustness and data integrity is to provide redundancy of the filesystem metadata. At a minimum, the filesystem should provide redundant copies of inodes and indirect blocks. If practical, the filesystem should also provide multiple copies of the directory data blocks. And if the implementation was flexible enough, it could provide optional redundancy for the user's data blocks as is available in ZFS.Another technology that is beginning to appear in the marketplace is key/value disks such as those recently released by Seagate. These disks provide objects up to one megabyte in size that are identified using a 64-bit key. The filesystem could be adapted to use these disks by using a one megabyte blocksize, which would dramatically reduce the amount of metadata information that it would need to maintain. The block numbers would be replaced with the 64-bit object keys, and the file content would be stored as the object value. The final fragment of the file could be stored in a smaller object, thus allowing the disk to manage disk fragmentation issues. ●

MARSHALL KIRK MCKUSICK writes books and articles, consults, and teaches classes on Unix- and BSD-related subjects. While at the University of California at Berkeley, he implemented the 4.2BSD fast file system and was the Research Computer Scientist at the Berkeley Computer Systems Research Group (CSRG), overseeing the development and release of 4.3BSD and 4.4BSD. He has twice been president of the board of the Usenix Association, is currently a member of the FreeBSD Foundation Board of Directors, a member of the editorial board of ***ACM Queue*** magazine and ***FreeBSD Journal***, a senior member of the IEEE, and a member of the Usenix Association, ACM, and AAAS. You can contact him via email at mckusick@mckusick.com.

FURTHER INFORMATION For those interested in learning more about the history of BSD, additional information is available from www.mckusick.com/history.

REFERENCES

[1] Apple, *Mac OS X Essentials, Chapter 9 Filesystem, Section 12 Resource Forks*, available from https://en.wikipedia.org/wiki/Resource_fork.

[2] D. Chamberlin & M. Astrahan. "A History and Evaluation of System R," *Communications of the ACM 24*(10), pp. 632–646. (October 1981)

[3] S. Chutani, O. Anderson, M. Kazar, W. Mason, & R. Sidebotham. "The Episode File System," *USENIX Association Conference Proceedings*, pp. 43–59. (January 1992)

CONTINUES NEXT PAGE

[4] G. Ganger & Y. Patt. "Metadata Update Performance in File Systems," *USENIX Symposium on Operating Systems Design and Implementation*, pp. 49–60. (November 1994)

[5] J. L. Griffin, J. Schindler, S. W. Schlosser, J. S. Bucy, & G. R. Ganger. "Timing-Accurate Storage Emulation," *Proceedings of the USENIX Conference on File and Storage Technologies*, pp. 75–88. (January 2002)

[6] R. Hagmann. "Reimplementing the Cedar File System Using Logging and Group Commit," *ACM Symposium on Operating Systems Principles*, pp. 155–162. (November 1987)

[7] J. S. Heidemann & G. J. Popek. "File-System Development with Stackable Layers," *ACM Transactions on Computer Systems* 12(1), pp. 58–89. (February 1994)

[8] D. Hendricks. "A Filesystem for Software Development," *USENIX Association Conference Proceedings*, pp. 333–340. (June 1990)

[9] D. Korn & E. Krell. "The 3-D File System," *USENIX Association Conference Proceedings*, pp. 147–156. (June 1989)

[10] C. R. Lumb, J. Schindler, & G. R. Ganger. "Freeblock Scheduling Outside of Disk Firmware," *Proceedings of the USENIX Conference on File and Storage Technologies*, pp. 275–288. (January 2002)

[11] A. Ma, C. Dragga, A. Arpaci-Dusseau, & R. Arpaci-Dusseau. "ffsck: The Fast File System Checker," *USENIX FAST '13 Conference*, available from www.usenix.org/conference/fast13/ffsck-fast-file-system-checker. (February 2013)

[12] M. K. McKusick. "Running fsck in the Background," *Proceedings of the BSDC on 2002 Conference*, pp. 55–64. (February 2002)

[13] M. K. McKusick. "Enhancements to the Fast Filesystem to Support Multi-terabyte Storage Systems," *Proceedings of the BSDC on 2003 Conference*, pp. 79–90. (September 2003)

[14] M. K. McKusick, W. N. Joy, S. J. Leffler, & R. S. Fabry. "A Fast File System for UNIX," *ACM Transactions on Computer Systems* 2(3), pp. 181–197, Association for Computing Machinery. (August 1984)

[15] M. K. McKusick & T. J. Kowalski. "fsck: The UNIX File System Check Program" in *4.4 BSD System Manager's Manual*, pp. 3:1–21, O'Reilly & Associates, Inc., Sebastopol, California. (1994)

[16] L. McVoy & S. Kleiman. "Extent-Like Performance from a UNIX File System," *USENIX Association Conference Proceedings*, pp. 33–44. (January 1991)

[17] J. Moran, R. Sandberg, D. Coleman, J. Kepecs, & B. Lyon. "Breaking Through the NFS Performance Barrier," *Proceedings of the Spring 1990 European UNIX Users Group Conference*, pp. 199–206. (April 1990)

[18] J. Ousterhout. "Why Aren't Operating Systems Getting Faster as Fast as Hardware?," *Summer USENIX Conference*, pp. 247–256. (June 1990)

[19] J. Pendry & M. K. McKusick. "Union Mounts in 4.4 BSD -Lite," *USENIX Association Conference Proceedings*, pp. 25–33. (January 1995)

[20] J. Pendry & N. Williams. "AMD: The 4.4 BSD Automounter Reference Manual" in *4.4 BSD System Manager's Manual*, pp. 13:1–57, O'Reilly & Associates, Inc., Sebastopol, California. (1994)

[21] H. Reiser. *The Reiser File System*, available from https://en.wikipedia.org/wiki/ReiserFS.

[22] T. Rhodes. *Free BSD Handbook, Chapter 3, Section 3.3 File System Access Control Lists*, available from http://www.FreeBSD.org/doc/en_US.ISO8859-1/books /handbook/fs-acl.html. (March 2014)

[23] M. Rosenblum & J. Ousterhout."The Design and Implementation of a Log-Structured File System," *ACM Transactions on Computer Systems* 10(1), pp. 26–52, Association for Computing Machinery. (February 1992)

[24] D. Rosenthal. "Evolving the Vnode Interface," *USENIX Association Conference Proceedings*, pp. 107–118. (June 1990)

[25] J. Schindler, J. L. Griffin, C. R. Lumb, & G. R. Ganger. "Track-Aligned Extents: Matching Access Patterns to Disk Drive Characteristics," Proceedings of the *USENIX Conference on File and Storage Technologies*, pp. 259–274. (January 2002)

[26] M. Seltzer, K. Bostic, M. K. McKusick, & C. Staelin. "An Implementation of a Log-Structured File System for UNIX," *USENIX Association Conference Proceedings*, pp. 307–326. (January 1993)

[27] M. Seltzer & K. Smith. "A Comparison of FFS Disk Allocation Algorithms," *Winter USENIX Conference*, pp. 15–25. (January 1996)

[28] M. Stonebraker. "The Design of the POSTGRES Storage System," *Very Large DataBase Conference*, pp. 289–300. (September 1987)

[29] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, & G. Peck. "Scalability in the XFS File System," *USENIX Association Conference Proceedings*, pp. 1–14. (January 1996)

[30] R. Watson. "Introducing Supporting Infrastructure for Trusted Operating System Support in Free BSD," *Proceedings of the BSDC on 2000 Conference*. (September 2000)

[31] R. Watson. "Trusted BSD: Adding Trusted Operating-System Features to Free BSD," *Proceedings of the Freenix Track at the 2001 USENIX Annual Technical Conference*, pp. 15–28. (June 2001)

[32] R. Watson, W. Morrison, C. Vance, & B. Feldman. "The Trusted BSD MAC Framework: Extensible Kernel Access Control for Free BSD 5.0," *Proceedings of the Freenix Track at the 2003 USENIX Annual Technical Conference*, pp. 285–296. (June 2003)

[33] M. Wu & W. Zwaenepoel. "eNVy: A Non-Volatile, Main Memory Storage System," *International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS), pp. 86–97. (October 1994)

SEE TEXT ONLY

BUILDING A FOUNDATION FOR SECURE, TRUSTED COMPUTING BASES

CHERI

CAPABILITY HARDWARE ENHANCED RISC INSTRUCTIONS

by Brooks Davis

BSD operating systems have been around since the 1980s, and the history of UNIX extends all the way back to 1969, but despite orders of magnitude growth in performance, storage, and memory capacity, we still use CPUs with computing models that are remarkably similar to the PDP-11 on which the early versions of UNIX were run. We have a flat virtual address space (now 64 bits instead of 16), TLB-based process virtualization of memory, and permissions at page granularity.

This model has many advantages and has served us well for many years, but it also suffers from serious disadvantages. Multiple classes of memory corruption bugs including buffer overflows are consistent sources of vulnerabilities despite years of work to address them. As a community, we employ many mitigation techniques such as address-space layout randomization (ASLR) and compartmentalization (also known as privilege-separation), but these have significant costs. ASLR has a non-negligible performance overhead, provides protection that is statistical not absolute, and certain common programming models such as pre-fork servers allow efficient automated attacks. Compartmentalization limits the impact of failures of other mitigations by confining risky parts of programs to environments where their privilege is limited. On compartmentalizing software, programmers transform programs into distributed computations, increasing context switch overhead and TLB pressure while converting single-address-space programs into distributed systems with all the attendant complexity. This means compartmentalization is commonly used only where it is a trivial match to the application's programming model (`uniq(1)`) or where vulnerabilities would be the most critical (`sshd(1)`, Chrome, or Firefox).

Developed under the auspices of the DARPA CRASH (Clean-slate design of Resilient, Adaptive, Secure Hosts) program, CHERI (Capability Hardware Enhanced RISC Instructions) is a hardware/software co-design project that challenges the assumptions we've made about hardware software interfaces for the last 40-plus years. We have developed extensions to the MIPS64 ISA that provide robust, fine-grained, hardware-enforced, process-scope memory *capabilities* that enforce object boundaries and permissions in a manner compatible with C pointers. We have further extended these capabilities to provide robust, efficient, in-process compartmentalization. Our extensions are implemented as a MIPS coprocessor allowing incremental deployment of CHERI features.

To develop and demonstrate these features, we have implemented an FPGA soft-core CPU, ported FreeBSD to it, and extended FreeBSD with support for CHERI capabilities [Woodruff]. We have added support to LLVM and Clang to use capabilities in C, both a hybrid mode where specially annotated pointers become capabilities and another where all pointers are capabilities [Chisnall]. Additionally, we have extended our FreeBSD port (CheriBSD) to support in-process compartmentalization of code and demonstrated the viability of the approach with the `tcpdump` program and `zlib` library [Watson].

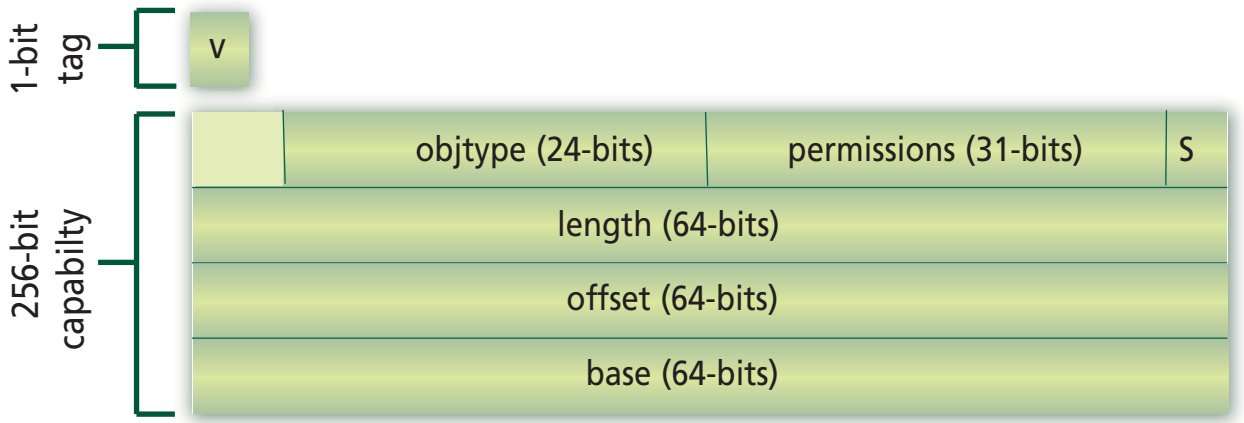
CHERI Capabilities

CHERI capabilities are unforgeable references to regions of virtual address space. They are stored in memory accompanied by a tag bit that verifies their validity (vs. some arbitrary arrangement of bits) and are manipulated in special capability registers. Capabilities contain a *base, length, offset* relative to the base, *permissions*, and a *type*. Instructions that manipulate capabilities may only shrink the region bounced by the base and length or reduce permissions. Attempts to increase the scope or permissions of a capability raise a hardware exception. Thus, capabilities provide monotonically decreasing rights.

All memory accesses are performed through a capability. This may occur directly via new capability-based load and store instruction or indirectly via the default data capability (`DDC`). When running capability-unaware or hybrid programs, `DDC` is set to a capability with rights to the whole address space. This allows all legacy load and store instructions to work as expected in hybrid or pure-MIPS binaries.

Because all memory accesses are via capabilities, the portion of address space that is reachable by a given thread is the transitive closure of the address space reachable by the set of capabilities in the register file. That is to say, all memory that can be accessed by a capability currently in the register file or via a capability that can be loaded from the memory the register file grants access to. Thus, compartmentalization can be achieved by transforming the contents of the register set. This is achieved with the `CCall` instruction, which takes a paired code and data capability having the same type, stores the current register contents, and sets up a new register set executing in the code capability and having access to argument registers and registers specified in the data capability. To exit a sandbox, a `CReturn` instruction restores the register set from a trusted stack.

In our primary prototype, capabilities are 256 bits and strongly aligned with a tag bit stored in a separate, inaccessible portion of `DRAM`. The current layout is shown in the figure below.



The primary overhead of using capabilities comes from the increased memory and especially cache footprint of code where pointers become capabilities. Pointer-intensive benchmarks have measurable overhead, but thus far real-world programs such as `tcpdump` show no significant penalty.

C Support

Unlike many past capability systems, we designed CHERI capabilities with the explicit goal of using them as C pointers. This has had considerable impact on our ISA and the contents of our capabilities. Most significantly, our capabilities include not only base and length, but offset, because real-world C programs often temporarily address values outside their allocated range.

Our initial focus on supporting capabilities in C was adding a new annotations `__capability` to pointers that we wished to restrict. For example, we annotated a version of `tcpdump` where we protected the packet pointer preventing out-of-bounds access from producing spurious results. The thousands of lines requiring annotation and the thousands of merge conflicts produced in the first import of a new `tcpdump` code base convinced us that a pure-capability mode was required for CHERI to be usable. We can currently compile nearly all C code in pure-capability mode and do so for most compartmentalized code. We continue to use hybrid mode in support code in `libc` that must be capability aware, and in transitions between MIPS64 and pure-capability code (usually around the edges of compartmentalized code).

Supporting CHERI capabilities as pointers also has a number of ABI subtleties that are beyond the scope of this article.

CheriBSD

To ensure that our ideas are truly viable, we have made it a goal from the start to run a real operating system and application stack on CHERI and to allow incremental adoption of CHERI features in software. To this end, we first brought up FreeBSD on CHERI adding support for those features required to support our prototype board without support for capabilities [Davis]. We separately added kernel support for running programs containing capabilities. This includes process startup, context switch code, signal handling, and debugging. The table below shows a breakdown of the modest set of changes required.

Component	Files Modified	Lines Added	Lines Removed
Headers	19	1,424	11
CHERI initialization	2	49	4
Context management	2	392	10
Exception handling	3	574	90
Memory copying	2	122	0
Virtual memory	5	398	27
Object capabilities	2	883	0
System calls	2	76	0
Signal delivery	3	327	71
Process monitoring/debugging	3	298	0
Kernel debugger	2	264	0

To support hybrid capability programs, we modified `libc` slightly to make memory manipulation function (`memcpy()`, `memmove()`, `qsort()`) capability aware to allow structures to contain capabilities. With these changes we continue to support unmodified MIPS64 binaries, and, in fact, most of our userspace programs are capability aware only in so far as it is easier (and potentially faster due to copying more bytes per instruction) to unconditionally use capability-aware memory manipulation functions. We have also provided capability aware variants of string and memory manipulation functions. For example, a capability away (and thus memory-safe) `strcpy_c()` implementation is shown below.

```
__capability char *
strcpy_c( __capability char * __restrict to,
     __capability const char * __restrict from)
{
         __capability char *save = to;

         for (; (*to = *from); ++from, ++to);

         return(save);
}
```

Due to the initially weak support for MISP64 in `clang` we modified the FreeBSD build system to allow select libraries to be built with our modified `clang` version while compiling the rest of the system with the

base `gcc`. We have also modified the build system to build pure-capability versions to all libraries similar to the way the normal build system builds 32-bit versions of libraries on 64-bit systems. This both allows us to more thoroughly test compiler support and to link unmodified libraries into compartmentalized sections of code that use pure-capability mode.

libcheri

Compartmentalization in CheriBSD is currently implemented in the `libcheri` library. It provides an interface to load sandbox classes and create sandbox objects. Sandbox objects are effectively mini-address spaces within a process—`libcheri` maps a region of address space, loads a compartmentalized object into it, and securely calls methods implemented by that object. In our current implementation, compartmentalized code is typically pure-capability code in order to take advantage of memory safety guarantees and to simplify passing of capabilities from outside the sandbox, but other models are possible. For example, a bit of wrapper code could allow an unmodified 32-bit library to run in a sandbox within a 64-bit program.

Conceptually, `libcheri` sandboxes are libraries with the added twist that multiple instances of each library can be instantiated and those instances can fail and be reset independently. This allows risky code such as `tcpdump` packet decoding or `zlib` decompression to be placed in a sandbox where failure has reduced consequences as the sandboxed code has no direct ability to make system calls and greatly reduced ability to impact the main process.

We have used `libcheri` compartmentalization to protect the main `tcpdump` process (often run as root!) from the packet dissection and printing code (handcrafted C to process untrustworthy and frequently corrupt data from the network). We are able to protect against a wide range of attack models from simple crashes to infinite loop-based denial of service attacks and even supply chain attacks where a packet triggers a bug in a malicious dissector. With these changes active, the impact of a crashing or denial of service bug is limited to the loss of formatted output for the given packet and a brief slowdown as we reload the sandbox instance.

In addition to protecting applications like `tcpdump` from their internals, we have also implemented library compartmentalization where we present an API- and ABI-compatible interface to a compartmentalized `zlib` library. This allows completely unmodified, dynamically linked `gzip`, `gif2png`, and similar programs to gain the benefits of a compartmentalized and memory safe `zlib` without even recompiling. This strategy of library compartmentalization allows the effort of compartmentalization to benefit as many consumers as possible with the least effort.

Contributions to FreeBSD

In the course of our work on CHERI and CheriBSD, we have contributed a number of changes back to FreeBSD. During our initial bring-up, we improved support for CFI flash devices, ported support for Flat Device Trees and the FreeBSD boot loaders to MIPS, and generally improved MIPS support. We have merged this work upstream as well as drivers for specific Altera and Terasic hardware on our Terasic DE4 reference board.

As we've worked on CHERI, our stricter interpretation of the C standard has found occasional correctness issues in code in FreeBSD. As we've found general issues, we've merged these changes as they will benefit both potential CHERI-based platforms and other memory safety systems like Intel's MXP once C language support is implemented.

We have also supported the development of QEMU user mode support to allow us to build packages for use on our FPGAs. Building packages on embedded boards like the EdgeRouter™ Lite is slow, but building them on a 100Mhz FPGA with slow I/O is completely impractical and so we've had to support the creation of new infrastructure.

Finally, should hardware implementations of CHERI emerge, CheriBSD stands ready as a reference platform and a base for mainline CHERI support in FreeBSD.

Future Work

We are currently exploring a new system call interface where system call argument pointers are capabili-

ties. This will allow us to run significant numbers of unmodified programs in pure-capability mode. Our hope is that we can build and safely run a FreeBSD userspace with full spatial memory in the near future. We expect to find a number of subtle bugs as we enforce bounds on such a large codebase.

Library compartmentalization is a powerful tool for improving the safety of diverse codebases. We have implemented a number of compartmentalization prototypes, but need to port a wider range of libraries. Doing so will help us determine the sorts of tools we need to develop to ease the process. We have already done some work in this area with our SOAAP project [Gudka], but have mostly focused on application compartmentalization. Library compartmentalization poses a similar but related set of issues. In some cases, it may be practical and advisable to implement process-based library compartmentalization for some libraries. Libraries with buffer-based interfaces such as `zlib` are ill-suited to this, but libraries with stream-oriented interfaces may achieve reasonable performance even in a process-oriented environment.

As cache pressure is one of the main performance impacts of CHERI, we are currently perusing a 128-bit compressed capability model for CHERI. Our experiments show this lowers the overhead significantly at the cost of a loss of granularity for large objects. We hypothesize that the loss of granularity largely maps to existing practices of rounding up allocations, but work is ongoing to confirm this.

Further Reading

This article only scratches the surface of our work on CHERI. Our three conference papers cover it in much greater detail and show the evolution of our ideas as we have developed more software support. *The CHERI capability model: Revisiting RISC in an age of risk* [Woodruff] covers the key memory safety properties of *CHERI. Beyond the PDP-11: Processor support for a memory-safe C abstract machine* [Chisnall] shows changes required to implement C on top of CHERI. Finally, *CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization* [Watson-Oakland] details our compartmentalization strategy.

Those interested in the nuts and bolts of the ISA may find *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture* [Watson-ISA] to be of interest.

Trying CHERI

We have released a QEMU implementation of CHERI which tracks our our CheriBSD, Clang, and LLVM progress on our Github repository https://github.com/CTSRD-CHERI/.

We have made our CHERI CPU FPGA implementation and snapshots of related software available as open source at http://chericpu.org. FPGA releases are infrequent and lag the current state of development. If you have a need for the FPGA version, please contact us.

BROOKS DAVIS is a Senior Software Engineer in the Computer Science Laboratory at SRI International and a Visiting Research Fellow at the University of Cambridge Computer Laboratory. He has been a FreeBSD user since 1994, a FreeBSD committer since 2001, and was a core team member from 2006 to 2012. Brooks earned a Bachelor's Degree in Computer Science from Harvey Mudd College in 1998. His computing interests include security, operating systems, networking, high-performance computing, and, of course, finding ways to use FreeBSD in all these areas. When not computing, he enjoys cooking, brewing, gardening, woodworking, blacksmithing, and hiking.

REFERENCES

[Chisnall] David Chisnall, Colin Rothwell, Brooks Davis, Robert N. M. Watson, Jonathan Woodruff, Simon W. Moore, Peter G. Neumann, and Michael Roe. "Beyond the PDP-11: Processor support for a memory-safe C abstract machine," Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS 2015), Istanbul, Turkey. (March 2015) http://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/201503-asplos2015-cheri-cmachine.pdf

[Davis] Brooks Davis, Robert Norton, Jonathan Woodruff, and Robert N. M. Watson. "Bringing Up MIPS," FreeBSD Journal. (January/February 2015)

[Gudka] Khilan Gudka, Robert N. M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. "Clean Application Compartmentalization with SOAAP," Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS 2015), Denver, Colorado. (October 2015) http://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/2015ccs-soaap.pdf

CONTINUES NEXT PAGE

REFERENCES CONTINUED

[Watson-Oakland] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. "CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization," Proceedings of the 36th IEEE Symposium on Security and Privacy ("Oakland"), San Jose, California. (May 2015) http://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/201505-oakland2015-cheri-compartmentalization.pdf

[Watson-programmers-guide] Robert N. M. Watson, David Chisnall, Brooks Davis, Wojciech Koszek, Simon W. Moore, Steven J. Murdoch, Peter G. Neumann, and Jonathan Woodruff. "Capability Hardware Enhanced RISC Instructions: CHERI Programmer's Guide," Technical Report UCAM-CL-TR-877, University of Cambridge, Computer Laboratory. (September 2015) Current CHERI programmer's guide http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-877.pdf

[Watson-ISA] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Jonathan Anderson, David Chisnall, Brooks Davis, Alexandre Joannou, Ben Laurie, Simon W. Moore, Steven J. Murdoch, Robert Norton, and Stacey Son. "Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture," Technical Report UCAM-CL-TR-876, University of Cambridge, Computer Laboratory. (September 2015) Current CHERI ISA specification http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-876.pdf

[Watson-BERI-software] Robert N. M. Watson, David Chisnall, Brooks Davis, Wojciech Koszek, Simon W. Moore, Steven J. Murdoch, Peter G. Neumann, and Jonathan Woodruff. "Blue spec Extensible RISC Implementation: BERI Software Reference," Technical Report UCAM-CL-TR-869, University of Cambridge, Computer Laboratory. (April 2015) http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-869.pdf

[Watson-BERI-hardware] Robert N. M. Watson, Jonathan Woodruff, David Chisnall, Brooks Davis, Wojciech Koszek, A. Theodore Markets, Simon W. Moore, Steven J. Murdoch, Peter G. Neumann, Robert Norton, and Michael Roe. "Blue spec Extensible RISC Implementation: BERI Hardware Reference," Technical Report UCAM-CL-TR-868, University of Cambridge, Computer Laboratory. (April 2015) http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-868.pdf

[Woodruff] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. "The CHERI capability model: Revisiting RISC in an age of risk," Proceedings of the 41st International Symposium on Computer Architecture (ISCA 2014), Minneapolis, Minnesota. (June 14–16, 2014) http://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/201406-isca2014-cheri.pdf

This paper is approved for public release; distribution is unlimited. It was developed with funding from the Defense Advanced Research Projects Agency (DARPA) under Contract FA8750-10-C-0237. The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies of the U.S. Department of Defense.

THE INTERNET NEEDS YOU

GET CERTIFIED AND GET IN THERE!

Go to the next level with  **BSD CERTIFICATION**

Getting the most out of BSD operating systems requires a serious level of knowledge and expertise

NEED AN EDGE?

BSD Certification can make all the difference.

Today's Internet is complex. Companies need individuals with proven skills to work on some of the most advanced systems on the Net. With BSD Certification

YOU'LL HAVE WHAT IT TAKES!

SHOW YOUR STUFF!

Your commitment and dedication to achieving the **BSD ASSOCIATE CERTIFICATION** can bring you to the attention of companies that need your skills.

BSDCERTIFICATION.ORG

Providing psychometrically valid, globally affordable exams in BSD Systems Administration

SEE TEXT ONLY

svn UPDATE

by Steven Kreuzer

A code freeze for 10.3-RELEASE has been in effect for a while now. By the time you read this column, it should be available. In this installment of svn update I intended to tell you about some of the new features and enhancements; however, I became so excited about the recent updates happening in 11-CURRENT that I decided to change this at the last minute. As many of you have been eagerly awaiting the arrival of 10.3, you can find a list of changes since 10.2-RELEASE in the stable/10 release notes (https://www.freebsd.org/relnotes/10-STABLE/relnotes/article.html). If you would like to help out by participating in the release cycle, I strongly encourage you to try out the 10.3-BETA builds which are now available for download. Be sure to report any regressions you might stumble upon.

Support for the RISC-V Instruction Set Architecture (ISA)

While i386 and amd64 are the most popular, FreeBSD supports numerous architectures that allow you to install your favorite operating system on exotic hardware ranging from embedded ARM micro controllers to large SPARC workhorses. A recent update added support for a new and completely open ISA called RISC-V that is being made available to anyone under a BSD license. RISC-V was originally developed at the University of California, Berkeley, to support computer architecture research and education and is now set to become a standard open architecture for industry implementations. FreeBSD is the first operating system to have bootable in-tree support for RISC-V, and because FreeBSD can trace its heritage back to Berkeley as well, it only seems natural that it should support this new and exciting architecture. https://svnweb.freebsd.org/changeset/base/295041

Kernel Support for the Vector-Scalar eXtension (VSX) Found on the POWER7 and POWER8

The IBM POWER architecture provides vector and vector-scalar operations through the VMX and VSX instruction sets, which are part of the version 2.06 POWER ISA. Support for this instruction set has been added to the FreeBSD/powerpc port and unifies the 32 64-bit scalar floating point registers with the 32 128-bit vector registers into a single bank of 64 128-bit registers. https://svnweb.freebsd.org/changeset/base/279189

The Xen Netfront Driver Gains Multiqueue Support

With continued support from Citrix Systems R&D, Xen on FreeBSD is actively being worked on and a recent update will help address one of the major sources of performance degradation in virtualized environments. In addition to undergoing a major refactoring, the ability to have multiple TX and RX queue pairs has been added to the netfront driver. Guest virtual machines that have heavy network workloads should see a significant improvement since one of the biggest costs when virtualizing I/O is how to efficiently allow multiple virtual machines to securely share access to single devices. https://svnweb.freebsd.org/changeset/base/294442

OpenSSH Has Been Upgraded to 7.1p2.

The latest version of OpenSSH has been committed to the tree. This new version addresses the recent Use Roaming security issue that is documented in CVE-2016-0777 and CVE-2016-0778 and also fixes an out-of-bound read access in the packet-handling code. In addition, further use of explicit_bzero has been added in various buffer handling code paths to guard against compilers aggressively doing dead-store removal. https://svnweb.freebsd.org/changeset/base/294496

HPN Has Been Removed from OpenSSH

HPN is a set of patches from the Pittsburgh Supercomputing Center that remove several bottlenecks in OpenSSH to improve network performance, especially on long- and high-bandwidth network links. Unfortunately, they provided limited usefulness, and it required quite a bit of effort to

maintain these patches in the tree. As a result, they have been removed along with the None cipher. While most users will not be affected by this change, those who are should switch to the openssh-portable port which still offers both patches and has HPN enabled by default. https://svnweb.freebsd.org/changeset/base/294325

OpenSSL Has Been Updated to Version 1.0.2f.

A new release of OpenSSL has been imported into the tree that addresses CVE-2016-0701 in which the `DH_check_pub_key` function does not ensure that prime numbers are appropriate for Diffie-Hellman (DH) key exchange, which makes it easier for remote attackers to discover a private DH exponent by making multiple handshakes with a peer that chose an inappropriate number. In addition, this new version also addresses CVE-2015-3197 in which OpenSSL does not prevent use of disabled ciphers, which makes it easier for man-in-the-middle attackers to defeat cryptographic protection mechanisms by performing computations on SSLv2 traffic, related to the `get_client_master_key` and `get_client_hello` functions. https://svnweb.freebsd.org/changeset/base/295009

rc_conf_files Can Be Redefined in `rc.conf(5)`

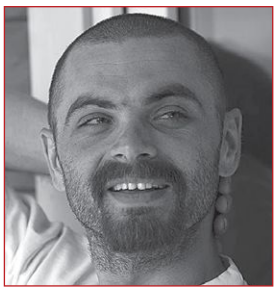
This is one change I think will be welcomed with open arms by system administrators who need to maintain an infrastructure of hundreds or thousands of FreeBSD machines. Being able to create a rc.conf file specific for a machine's role and introduce that configuration into your environment without having to perform edits to `/etc/rc.conf` makes it easier to tell your junior admin to move this knowledge into a configuration management system and allows them to do so without having to learn the arcane syntax or templating required by such a system. While this change may appear to be minor, any steps that FreeBSD developers take to lower the barrier to entry and make the lives of the folks who are just getting started easier is a step in the right direction to making sure that FreeBSD will continue to see widespread adoption across all industries. https://svnweb.freebsd.org/changeset/base/295342

STEVEN KREUZER is a FreeBSD Developer and Unix Systems Administrator with an interest in retro-computing and air-cooled Volkswagens. He lives in Queens, New York, with his wife, daughter, and dog.

SEE TEXT ONLY

this month In FreeBSD

BY DRU LAVIGNE



An Interview with Gleb Smirnoff

Over the next few issues, we'll be taking a closer look at some of the new features making their way into the 2016 releases and the developers behind those features.

This month we chat with Gleb Smirnoff, a member of the FreeBSD core, release engineering, and security teams. He is also a senior software developer at Netflix Inc.

Q Tell us a bit about yourself. How did you get started with FreeBSD and what is your involvement with the FreeBSD Project?

A I started using FreeBSD back in 2000, when I was 18. At the university dormitory, we were building a LAN and we used FreeBSD for routing, Internet sharing, and web services. Quite quickly I went from documentation to sources. I was fascinated with `netgraph(4)`. In 2004, I received committer access, and from that point all my life is coupled with FreeBSD.

Q You have been working on the next generation of `sendfile(2)`, which is typically used to increase web server performance. Please provide an overview of the original `sendfile(2)` implementation. Does the new `sendfile(2)` address any of its shortcomings and what performance improvement does it provide?

A The original sendfile was quite straightforward. It locked the socket buffer to protect it from any other writes, grabbed memory for the requested data, and filled the memory with data taken from a disk. Once data was read from disk, it put the data into the socket buffer, unlocked the buffer, and returned.

The shortcoming is that it takes time to read data from a disk. As an example, let's assume it takes 5 milliseconds to read the requested data from a disk. This means we can do only 200 requests per second. To do more requests, we need to spawn extra threads or processes for the sendfile job. Actually, we will end up creating extra contexts just for waiting on disk.

A high-performance web server, such as nginx, is written as an event dispatcher which puts all of its sockets and file descriptors into non-blocking mode and then polls them. As a descriptor becomes available for a read or write, the web server sends or reads data and goes to the next one. This works well when system calls are fast. With modern demands, the original sendfile became a bottleneck, particularly in regard to number of connections and throughput.

This was a well-known problem even a decade ago. FreeBSD used a special flag, `SF_NODISKIO`, to tell sendfile to avoid reading from disk if data is not cached, and to return a special error code immediately. The web server then pre-cached the data with a call to `aio_read(4)` and retried the sendfile. Although there is a lot of extra action in place, and there is no guarantee that data will wait in cache for sendfile, this approach worked much better than letting the original sendfile block on disk.

We decided to make it even better. The idea is that sendfile will not wait for disk to read the data and will return immediately. This means that the web server doesn't stall for several milliseconds and can go forward working with the next descriptor. In short, that is the whole idea, but the implementation is not as simple as the description.

Q Were substantial changes required to the FreeBSD kernel or any of its subsystems to implement the new sendfile(2)? Did you come across any unexpected bugs during the implementation?

A Yes, the changes required were substantial. First, we needed to create an asynchronous interface in the kernel to read the data. The original `sendfile(2)` talked to disk similar to a `read(2) syscall`, using the `VOP_READ` filesystem operation. This was already wrong, since the interface is designed to copy out data to userland, which in principle is what sendfile is meant to avoid. The original sendfile pre-wired

pages that correspond to the data read, then ran `VOP_READ` with copy out to nowhere. As a side effect, that made pages paged in, and thus ready to be sent to a socket. Thus, we decided not to delve into the `VOP_READ_ASYNC` interface, but to instead use `VOP_GETPAGES`, which is designed exactly for bringing individual pages into memory. We implemented `VOP_GETPAGES_ASYNC` and built the new sendfile around it.

Second, substantial change affected socket buffers as we put data into a socket, but the data isn't ready yet. Since the pages aren't yet read from a disk, we can't send them. But they must occupy their place in the socket to preserve correct sequencing of data in a socket. When accounting for socket buffer limits, we also must account for that data. This all required bringing a notion of not ready data in socket buffers and functions to write it and to later activate it.

Speaking of bugs: of course, during experiments we found a lot of them. Who doesn't? An interesting one was an arithmetic bug in the `vnode_pager_haspage()` that was inherited from pre-FreeBSD times, when it suggested that it can read beyond the end of a file. We were the first to extensively use this function, which is why we found the bug after 20 years. It was fixed in FreeBSD `commit r282426`.

Q The new sendfile(2) was a joint effort between NGINX Inc. and Netflix. Why was FreeBSD chosen as the reference implementation and are there plans to port this system call to other operating systems such as Linux?

A FreeBSD is used in the heart of Netflix OpenConnect CDN[1], which streams data to Netflix customers all over the world, being the world's largest source of traffic on the Internet. So we were not building a reference implementation of the new sendfile just for fun or as a thought experiment; rather, we were improving the OpenConnect software, so that a single server can serve more data. This resulted in improving FreeBSD itself.

There are no plans for porting this to other operating systems on our side. We are focused on FreeBSD.

Q As a developer at Netflix, you have the unique opportunity to work with a CDN designed to push massive amounts of Internet traffic using FreeBSD. Were other FreeBSD improvements needed to manage the scale required by Netflix, and does Netflix upstream most of its improvements for the benefit of the FreeBSD community?

A Yes, we have made a lot of extra changes to FreeBSD to serve our traffic. And yes, we are trying to upstream all of them. The process isn't easy though. The quality standards for open-source code are actually higher than for production code. In production, you care only about the architecture you are running on, and you don't care about others. You care only about your workload, and you care only about those parts of an operating system that you use. To upstream something, we need to address all other platforms, workloads, and possible users of APIs and standards. At the same time, adopting our code to meet open-source standards must not degrade performance for our case. Sometimes satisfying both internal and open-source demands at the same time appears difficult.

Nevertheless, the process of upstreaming our improvements goes on. Follow commits from emax@, gallatin@, glebius@, imp@, lstewart@, rrs@, and scottl@ to see what is going on for FreeBSD 11 and 12.

Q Are you working on any other interesting projects?

A Right now, I don't have any big projects in my queue, but there is a lot of interesting stuff from my colleagues at Netflix.

For example, there is `SSL_sendfile()`, built on top of the new sendfile. The idea is that once a TLS session is negotiated, the web server gives the session key to the kernel, and then can use sendfile on a TLS socket. The implementation requires two stages of asynchronous data processing: first, disk read, and second, encryption. Only then is the data in a socket activated.

There is also an I/O scheduler and massive improvements to the VM subsystem, to TCP, to storage drivers, and to NIC drivers. These topics actually deserve separate articles, so I won't go into detail and leave that to their respective authors. ●

[1] https://www.nginx.com/blog/why-netflix-chose-nginx-as-the-heart-of-its-cdn/

Dru Lavigne is a Director of the FreeBSD Foundation and Chair of the BSD Certification Group.

SEE TEXT ONLY

PORTSreport

by Frederic Culot

The Level of Activity during the January–February period was very high on the problem-fixing front. During this period, we were also pleased to welcome miwi@ back to the port management team and to bring three new or returning committers onboard. Also, some major ports were updated, which may require caution when upgrading, as described below.

NEW PORTS COMMITTERS AND SAFEKEEPING

Three commit bits were taken in for safekeeping during the first two months of the year: mmoll, milki, and brian. On the other hand, two new commits were awarded. The first goes to Olivier Cochard-Labbe, who was already well-known to the community as the creator of FreeNAS and the BSD Router project (BSDRP). The second commit bit was awarded to Christoph Moench-Tegeder, who submitted some major patches to Chromium, Firefox, and Thunderbird, together with almost 200 problem reports since 2005. In February, we were also delighted to reinstate Dima Panov (fluffy@)'s commit bit, which was taken into safekeeping last September after 18 months of inactivity.

STATS

February figures were not yet available at the time of writing, but based on the ones for January, the numbers are encouraging: more committers worked on the ports tree than during the last period (128, an increase of more than 10%), which led to an increase of almost 20% in the number of commits. But more significant is the number of problem reports that were closed in January, which increased by more than 30%. This is very stimulating, and let's hope all the developers will manage to keep up the pace!

INFRASTRUCTURE

A lot of work was done by bdrewery@ and antoine@ to improve the package building and testing infrastructure. This involved upgrading the system on the machines hosting those services as well as moving the machine that hosts the pkg-status service (https://pkg-status.freebsd.org). Some work was also done to improve the performance of the build system.

A CHANGE in the Management Team

In January, a vote was held to bring miwi back into the portmgr team. The outcome was unanimously positive, making miwi the sixth portmgr member. We have missed miwi since the end of 2014 when he had to step down from his duties to allow time for his growing family and real job. We are all pleased to welcome him back to our ranks! That being said, if you wish to know more about the portmgr team and its duties, you can start with the link https://www.freebsd.org/portmgr/.

IMPORTANT PORTS UPDATES

As always, many exp-runs (around 30) were run by antoine@, to check whether major ports updates are safe or not. Among those important updates, we can mention the following highlights:

- Ruby 2.3 was added to the tree and default version set to 2.2
- ruby-gens updated to 2.5.1
- clang updated to 3.8.0
- Qt5 updated to 5.5.1
- setuptools updated to 20.0
- Gnome updated to 3.18

As usual, more information is available in the /usr/ports/UPDATING file, which should be read carefully before upgrading major ports.

Having completed his PhD in Computer Modeling applied to Physics, FREDERIC CULOT has worked in the IT industry for the past 10 years. During his spare time he studies business and management and just completed an MBA. Frederic joined FreeBSD in 2010 as a ports committer, and since then has made around 2,000 commits, mentored six new committers, and now assumes the responsibilities of portmgr-secretary.

SEE TEXT ONLY

conference REPORT

by Rodrigo Osorio

FOSDEM (https://fosdem.org)

FOSDEM, one of the major free and noncommercial FOSS events in Europe, took place in Brussels, Belgium, on the weekend of January 30 and 31, 2016.

Every year, this event attracts more than 5,000 hackers and about 600 lectures. As part of the program, FOSS projects can submit a proposal for a developer room (devroom). The BSD devroom was held on Sunday, and I was in charge of the organization of that devroom. This year, a FreeBSD devsummit was also organized for the Saturday.

FOSDEM officially starts on Friday afternoon with the "Beer Event" at the Delirium Café. Since this place is usually crowded, it's more convenient to meet others in one of the surrounding bars. During this night, Brussels becomes a magical city where you can unexpectedly meet hundreds of software developers on every street corner and share a beer together.

The FreeBSD devsummit was held in central Brussels, close to the central station and the "Grande Place." The hottest topic was the effort to archive reproducible builds for base and the ports (https://wiki.freebsd.org/ReproducibleBuilds). In the afternoon, some attendees moved to the FOSDEM venue to see gnn@ talk about networking benchmarks.

Saturday is also the day for organized dinners. We had one with devsummit attendees in an elegant restaurant in one of the oldest European glazed shopping arcades in Brussels.

Sunday was the longest day. Devroom managers must arrive by 8 a.m., collect the recording material, carry it to the devroom, plug in everything, and be ready to operate at 9 a.m. Hopefully, the heaviest equipment is already in the room, left by the Saturday folks, and this year the FOSDEM video team did the cabling for us.

Talks start at 9 a.m. and finish around 5 p.m. with 5 minute breaks between each talk. All of the talks were very good and a lot of people attended the BSD devroom session. Some of the notable presentations included:

The EdgeBSD project talk, which started the day, turned out to be the single NetBSD-related talk.

Allan Jude's ZFS talk forced us to put the "full" sign on the door before the talk began, and many people had to stand to attend this presentation.

François Revol from the Haiku project was the only non-BSD speaker, but he discussed how BSD code and concepts improve the Haiku project.

I want to thank everyone for their help and support in the organization of the BSD devroom, especially Roger Pau Monné, Marius Nuennerich, and Baptiste Daroussin.

Hopefully next year we will see the same quality of talks, more diversity in the BSDs, and possibly a two-day devroom. ●

RODRIGO OSORIO is a software engineer working at Satcom1, an aeronautical satellite services provider. He has been a FreeBSD port committer since 2014 and a longtime BSD enthusiast and advocate.

meeting REPORT

by Marcelo Araujo

PortsCamp

in Taipei/Taiwan

January 15, 2016

This, the first PortsCamp held in Taipei's HackerSpace, had 27 attendees, including five FreeBSD committers: araujo@, lwhsu@, sunpoet@, kevlo@, and ijiliao@. The event had one sponsor, Gandi, which generously provided the venue as well as drinks and pizza. I made FreeBSD stickers that were shared with everyone.

Our event featured two presentations: in the first, Thomas Kuiper, general manager of Gandi's Asian subsidiary introduced how Gandi is using FreeBSD in its cloud platform. He also provided trial accounts to anyone who would like to try the FreeBSD images in their IaaS environment (Gandi: http://.www.gandi.net).

I gave the second presentation, which was an overview of the FreeBSD Project in which I explained how to contribute and become a FreeBSD committer. Slides from my presentation are at http://www.slideshare.net/araujobsd/portscamp-taiwan.

I also provided a server with `bhyve(8)` with the latest FreeBSD-HEAD, ports tree and poudriere(8), which the attendees could access by ssh and then experiment with FreeBSD ports. Providing the server was a very good idea, as the environment was ready to be used, saving a lot of setup time. The server was an i7 with 16G of ram. It was hosting 25VMs, and the performance was pretty good.

We had 193 commits in our ports tree with the history log: Sponsored by: PortsCamp Taiwan.

The attendees spent most of their time understanding how to use `poudriere(8)`. Some were already working on projects. For example, one person was testing glusterfs patches. Another was a dentist working on translating documentation into Chinese. All committers were helpful and spent most of their time answering questions and chatting with everyone.

We had a very good gathering, and we are planning to have a second PortsCamp in another city in Taiwan (Hsinchu). It will probably be held at National Taiwan University right after the AsiaBSDCon in March of this year. We may change the approach of PortsCamp to a working group format which we believe would be more productive.

I would like to give a special thanks to Mose for offering great support before and during the PortsCamp.

Here are some other blogs and posts about PortsCamp:
by kevlo: http://wp.me/p1J5gU-2v
by miwi: http://miwi.cc/2016/01/aiwan-portscamp/

MARCELO ARAUJO is a FreeBSD developer working mainly on ports and recently on kernel-related features. He is a Brazilian guy who lives in Taiwan and outside of the free software world. Marcelo likes motorcycles, hiking, and beer.

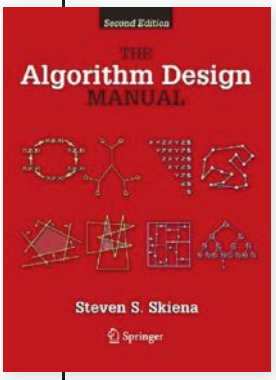
SEE TEXT ONLY

BOOKreview

by Joseph Kong

The Algorithm Design Manual

2nd Edition by Steven S. Skiena

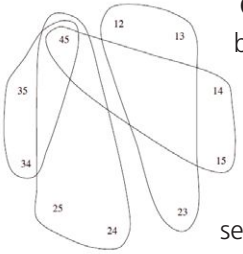


Publisher .....*Springer; 2nd edition (July 2008)*
Print List Price *$89.95*
eBook List Price.....*$69.99*
ISBN-10.....*1848000693*
ISBN-13.....*978-1848000698*
Pages *730*

This is one of the best books I've read in the past year. It's a steady tour through the world of algorithms with just the right amount of handholding (definitely not for novice programmers though).

The Algorithm Design Manual is divided into two parts. The first part, Chapters 1–10, describes the fundamentals of algorithm design, including data structures, dynamic programming, backtracking, modeling, and so on. Each chapter also contains several "war stories," which detail Skiena's experience with real-world problems. These stories are quite entertaining and help lighten the mood on some rather dense material.

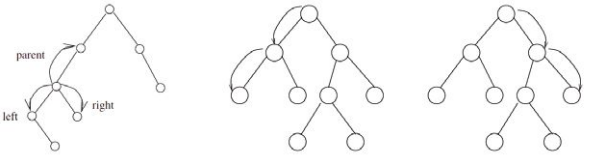
The second part, Chapters 11–19, is a catalog of algorithmic resources (for example, data structures, implementations, and so on) and is intended for browsing and reference. The idea is that instead of solving algorithmic problems from scratch, you use an existing one as a starting point (or the solution).



Chapter 1, Introduction to Algorithm Design, begins by describing what an algorithm is before going into detail about modeling, which is the art of breaking down a problem into one or more well-understood problems (whose solution can be found in the second part of this book).

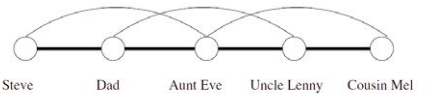
Chapter 2, Algorithm Analysis, describes algorithm efficiency and introduces the "big Oh" notation. Here, I found Skiena's explanation of "big Oh" to be quite good—definitely better than a lot of other texts.

Chapter 3, Data Structures, describes arrays, linked lists, stacks, queues, dictionaries, binary trees, priority queues, and hash tables all in the context of algorithm design. I found this chapter to be a very good refresher course.



Chapter 4, Sorting and Searching, stresses how sorting can be applied to solve other problems. The concepts of binary search and divide-and-conquer are also described. Several algorithms are detailed here, including heapsort, mergesort, quicksort, and distribution sort.

Chapter 5, Graph Traversal, begins by describing the different types of graphs (for example, directed, undirected, weighted, unweighted, and so on) and data structures used for graphs (that is, adjacency matrixes and adjacency lists) before describing graph traversal techniques (that is, breadth-first search and depth-first search).

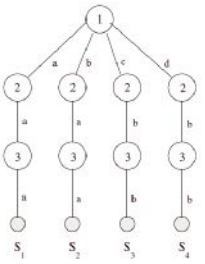


Chapter 6, Weighted Graph Algorithms, is a continuation of Chapter 5, which delves into weighted graphs, including discussions about spanning trees. Here, I found the war story about typing text via a telephone—the old-school way, where inputting 2 meant either the letter A, B, or C—and how to identify the correct letter via context to be rather entertaining and informative.

Chapter 7, Combinatorial Search and Heuristic Methods, describes how to systematically iterate through all the possible solutions (or configura-

tions) of a problem (or search space). The concepts of backtracking and search pruning are described here. I particularly enjoyed Skiena's walk-through on how to solve any Sudoku problem, as it really highlighted the power of backtracking and search pruning.

Chapter 8, Dynamic Programming, as the title suggests, describes dynamic programming. I found Skiena's explanation of the subject to be quite clear and easy to understand.



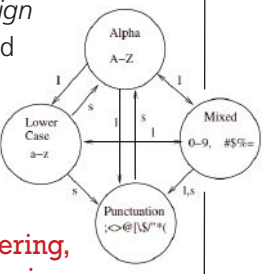
Chapter 9, Intractable Problems and Approximation Algorithms, introduces techniques for proving that no efficient algorithm exists for a particular problem. The idea is that if you can prove that searching for an efficient algorithm is pointless, you can focus your efforts elsewhere.

Chapter 10, How to Design Algorithms, is simply a four-page checklist to keep in mind when designing algorithms. While not much of a chapter, I greatly appreciated this list.

Chapters 11–19 contain a catalog of algorithmic resources detailing data structures, numerical problems, combinatorial problems, graph problems, computational geometry, and set and string problems. This portion of the book is affectionately known as "The Hitchhiker's Guide to Algorithms."

If I had to say something negative about this book, it'd be that it's dry and dense in parts—it's definitely not lazy Sunday afternoon reading. But these are minor nits.

The bottom line is that T*he Algorithm Design Manual* is a solid book and if you haven't read it yet, you should. ●



JOSEPH KONG is a self-taught computer enthusiast who dabbles in the fields of exploit development, reverse code engineering, rootkit development, and systems programming (FreeBSD, Linux, and Windows). He is the author of the critically acclaimed *Designing BSD Rootkits* and *FreeBSD Device Drivers*. For more information about Joseph Kong visit www.thestackframe.org or follow him on Twitter @JosephJKong.

2016 Events Calendar

The following BSD-related conferences will take place in April–June 2016. More information about these events, as well as local user group meetings, can be found at **www.bsdevents.org**.

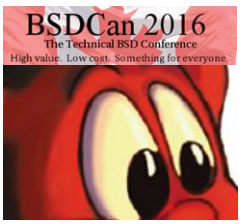
Hackathon • April 22–24 • Essen, Germany

https://wiki.freebsd.org/DevSummit/201604 • The second annual Hackathon and DevSummit will be held at the LinuxHotel in Essen. This event is open to both FreeBSD committers and contributors. There is a nominal charge for food and lodging.

LinuxFest NorthWest • April 23 & 24 • Bellingham, WA

http://linuxfestnorthwest.org/2016 • This is the 17th year for this annual, community-based conference. This event is free to attend and always has at least one BSD booth in the expo area and at least one BSD-related presentation.

BSDCan • June 8–11 • Ottawa, ON



https://www.bsdcan.org/2016/ • The 13th annual BSDCan will take place in Ottawa, Canada. This popular conference appeals to a wide range of people, from extreme novices to advanced developers of BSD operating systems. The conference includes a Developer Summit, Vendor Summit, Doc Sprints, tutorials, and presentations. The BSDA certification exam and the beta exam for the BSDP will also be available during this event.

SouthEast LinuxFest • June 10–12 • Charlotte, NC

http://www.southeastlinuxfest.org/ • The eighth annual SouthEast LinuxFest is a community event for anyone who wants to learn more about open-source software. There will be several FreeBSD-related presentations and a FreeBSD booth in the expo area.